Representing Action and Change by Logic Programs

Michael Gelfond
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968

Vladimir Lifschitz
Department of Computer Sciences
and Department of Philosophy
University of Texas at Austin
Austin, TX 78712

Abstract

We represent properties of actions in a logic programming language that uses both classical negation and negation as failure. The method is applicable to temporal projection problems with incomplete information, as well as to reasoning about the past. It is proved to be sound relative to a semantics of action based on states and transition functions.

1 Introduction

This paper extends the work of Eshghi and Kowalski [6], Evans [7] and Apt and Bezem [1] on representing properties of actions in logic programming languages with negation as failure.

Our goal is to overcome some of the limitations of the earlier work. The existing formalizations of action in logic programming are adequate for only the simplest kind of temporal reasoning—"temporal projection." In a temporal projection problem, we are given a description of the initial state of the world, and use properties of actions to determine what the world will look like after a series of actions is performed. Moreover, the existing formalizations can be used for temporal projection only in the cases when the given description of the initial state is complete. The reason for that is that these formalizations use the semantics of logic programming which automatically apply the "closed world assumption" to each predicate.

We are interested here in temporal reasoning of a more general kind, when the values of some fluents[1] in one or more situations are given, and the goal is to derive other facts about the values of fluents. Besides temporal projection, this class of reasoning problems includes, for instance, the cases when we want to use information about the current state of the world for answering questions about the past.[2]

The view of logic programming accepted in this paper is strictly declarative. The adequacy of a representation of a body of knowledge in a logic programming language means, to us, adequacy with respect to the declarative semantics of that language. In fact, the language of "extended logic programs" used in this paper is a subset of the system of default logic from [30], and our work can be viewed as a development of the approach to temporal reasoning based on nonnormal defaults [25]. The possibility of using the logic programs proposed in this paper for the automation of temporal reasoning, based on program transformations and the XOLDTNF metainterpreter [4], is demonstrated in the forthcoming paper [20].

Two parts of this paper may be of more general interest.

First, we introduce here a simple declarative language for describing actions, called $\mathcal{A}$. Traditionally, ideas on representing properties of actions in classical logic or nonmonotonic formalisms are explained on specific examples, such as the "Yale shooting problem" from [14]. Competing

[1]A *fluent* is something that may depend on the situation, as, for instance, the location of a moveable object [24]. In particular, *propositional* fluents are assertions that can be true or false depending on the situation.

[2]One possible way to represent reasoning about the past is to treat it as fundamentally different from temporal projection, and interpret it as "explanation" and "abduction" [33]. Our approach is more symmetric; we treat both forms of reasoning as deductive.

approaches are evaluated and compared in terms of their ability to handle such examples. We propose to supplement the use of examples by a different method. A particular methodology for representing action can be formally described as a translation from $\mathcal{A}$, or from a subset or a superset of $\mathcal{A}$, into a "target language"—for instance, into a language based on classical logic or on circumscription, or into a logic programming language.

Our method for describing properties of actions in logic programming is presented here as a translation from $\mathcal{A}$ into the language of extended logic programs, and its soundness is the main technical result of the paper. A counterexample is given showing that the translation is incomplete. A possible way of achieving completeness is discussed in the last section.

Second, the proof of the main theorem depends on a relationship between stable models [11] and signings [18], that may be interesting as a part of the general theory of logic programming.

The language $\mathcal{A}$ is introduced in Section 2, and Section 3 is a brief review of extended logic programs. Our translation from $\mathcal{A}$ into logic programming is defined in Section 4, and the soundness theorem is stated in Section 5. Section 6 contains the lemmas that relate stable models to signings, and in Section 7 the proof of the soundness theorem is presented.

2 A Language for Describing Actions

A description of an action domain in the language $\mathcal{A}$ consists of "propositions" of two kinds. A "value proposition" specifies the value of a fluent in a particular situation—either in the initial situation, or after performing a sequence of actions. An "effect proposition" describes the effect of an action on a fluent.

We begin with two disjoint nonempty sets of symbols, called *fluent names* and *action names*. A *fluent expression* is a fluent name possibly preceded by $\neg$. A *value proposition* is an expression of the form

$$F \textbf{ after } A_1; \ldots ; A_m, \tag{1}$$

where F is a fluent expression, and $A_1, \ldots, A_m$ $(m \geq 0)$ are action names. If $m = 0$, we will write (1) as

$$\textbf{initially } F.$$

An *effect proposition* is an expression of the form

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n, \tag{2}$$

where A is an action name, and each of $F, P_1, \ldots, P_n$ $(n \geq 0)$ is a fluent expression. About this proposition we say that it *describes the effect of A on F*, and that $P_1, \ldots, P_n$ are its *preconditions*. If $n = 0$, we will drop **if** and write simply

$$A \textbf{ causes } F.$$

A *proposition* is a value proposition or an effect proposition. A *domain description*, or simply *domain*, is a set of propositions (not necessarily finite).

Example 1. The Fragile Object domain, motivated by an example from [32], has the fluent names *Holding*, *Fragile* and *Broken*, and the action *Drop*. It consists of two effect propositions:

$$\begin{array}{l} \mathit{Drop} \textbf{ causes } \neg \mathit{Holding} \textbf{ if } \mathit{Holding}, \\ \mathit{Drop} \textbf{ causes } \mathit{Broken} \textbf{ if } \mathit{Holding}, \mathit{Fragile}. \end{array}$$

Example 2. The Yale Shooting domain, motivated by the example from [14] mentioned above, is defined as follows. The fluent names are *Loaded* and *Alive*; the action names are *Load*, *Shoot* and *Wait*. The domain is characterized by the propositions

$$\begin{array}{l} \textbf{initially } \neg \mathit{Loaded}, \\ \textbf{initially } \mathit{Alive}, \\ \mathit{Load} \textbf{ causes } \mathit{Loaded}, \\ \mathit{Shoot} \textbf{ causes } \neg \mathit{Alive} \textbf{ if } \mathit{Loaded}, \\ \mathit{Shoot} \textbf{ causes } \neg \mathit{Loaded}. \end{array}$$

Example 3. The Murder Mystery domain, motivated by an example from [2], is obtained from the Yale Shooting domain by substituting

$$\neg \mathit{Alive} \textbf{ after } \mathit{Shoot}; \mathit{Wait} \tag{3}$$

for the proposition **initially** $\neg$*Loaded*.

Example 4. The Stolen Car domain, motivated by an example from [16], has one fluent name *Stolen* and one action name *Wait*, and is characterized by two propositions:

$$\textbf{initially } \neg Stolen,$$
$$Stolen \textbf{ after } Wait; Wait; Wait.$$

To describe the semantics of $\mathcal{A}$, we will define what the "models" of a domain description are, and when a value proposition is "entailed" by a domain description.

A *state* is a set of fluent names. Given a fluent name F and a state σ, we say that F *holds* in σ if $F \in \sigma$; $\neg F$ *holds* in σ if $F \notin \sigma$. A *transition function* is a mapping Φ of the set of pairs (A, σ), where A is an action name and σ is a state, into the set of states. A *structure* is a pair (σ_0, Φ), where σ_0 is a state (the *initial state* of the structure), and Φ is a transition function.

For any structure M and any action names $A_1, \ldots, A_m$, by $M^{A_1;\ldots;A_m}$ we denote the state

$$\Phi(A_m, \Phi(A_{m-1}, \ldots, \Phi(A_1, \sigma_0) \ldots)),$$

where Φ is the transition function of M, and σ_0 is the initial state of M. We say that a value proposition (1) is *true* in a structure M if F holds in the state $M^{A_1;\ldots;A_m}$, and that it is *false* otherwise. In particular, a proposition of the form **initially** F is true in M iff F holds in the initial state of M.

A structure (σ_0, Φ) is a *model* of a domain description D if every value proposition from D is true in (σ_0, Φ), and, for every action name A, every fluent name F, and every state σ, the following conditions are satisfied:

(i) if D includes an effect proposition describing the effect of A on F whose preconditions hold in σ, then $F \in \Phi(A, \sigma)$;

(ii) if D includes an effect proposition describing the effect of A on $\neg F$ whose preconditions hold in σ, then $F \notin \Phi(A, \sigma)$;

(iii) if D does not include such effect propositions, then $F \in \Phi(A, \sigma)$ iff $F \in \sigma$.

It is clear that there can be at most one transition function Φ satisfying conditions (i)–(iii). Consequently, different models of the same domain description can differ only by their initial states. For instance, the Fragile Object domain (Example 1) has 8 models, whose initial states are the subsets of

$$\{Holding, Fragile, Broken\};$$

in each model, the transition function is defined by the equation

$$\Phi(Drop, \sigma) = \begin{cases} \sigma \setminus \{Holding\} \cup \{Broken\}, & \text{if } Holding, Fragile \in \sigma, \\ \sigma \setminus \{Holding\}, & \text{otherwise.} \end{cases}$$

A domain description is *consistent* if it has a model, and *complete* if it has exactly one model. The Fragile Object domain is consistent, but incomplete. The Yale Shooting domain (Example 2) is complete; its only model is defined by the equations

$$\begin{array}{l} \sigma_0 = \{Alive\}, \\ \Phi(Load, \sigma) = \sigma \cup \{Loaded\}, \\ \Phi(Shoot, \sigma) = \begin{cases} \sigma \setminus \{Loaded, Alive\}, & \text{if } Loaded \in \sigma, \\ \sigma, & \text{otherwise,} \end{cases} \\ \Phi(Wait, \sigma) = \sigma. \end{array}$$

The Murder Mystery domain (Example 3) is complete also; it has the same transition function as Yale Shooting, and the initial state $\{Loaded, Alive\}$. The Stolen Car domain (Example 4) is inconsistent.

A value proposition is *entailed* by a domain description D if it is true in every model of D. For instance, Yale Shooting entails

$$\neg Alive \textbf{ after } Load; Wait; Shoot.$$

Murder Mystery entails, among others, the propositions

$$\textbf{initially } Loaded$$

and

$$\neg Alive \textbf{ after } Wait; Shoot.$$

Note that the last proposition differs from (3) by the order in which the two actions are executed. This example illustrates the possibility of reasoning about alternative "possible futures" of the initial situation.

The language $\mathcal{A}$ is adequate for formalizing several interesting domains. Note that the domains from Examples 1–3, although very simple, have been actually proposed in the literature as counterexamples demonstrating the inadequacy and limitations of some earlier approaches to formalizing action. In many respects, however, the expressive power of $\mathcal{A}$ is rather limited. Some ways of extending $\mathcal{A}$ are mentioned in Section 8.

The entailment relation of $\mathcal{A}$ is nonmonotonic, in the sense that adding an effect proposition to a domain description D may nonmonotonically change the set of propositions entailed by D. (This cannot happen when a value proposition is added.) For this reason, a modular translation from $\mathcal{A}$ into another declarative language (that is, a translation that processes propositions one by one) can be reasonably adequate only if this other language is nonmonotonic also.

3 Extended Logic Programs

Representing incomplete information in traditional logic programming languages is difficult. Given a ground query, a traditional two-valued logic programming system can produce only one of two answers, *yes* or *no*; it will never tell us that the truth value of the query cannot be determined on the basis of the information included in the program.

Extended logic programs, introduced in [12], are, in this sense, different. The language of extended programs distinguishes between negation as failure *not* and classical negation $\neg$. The expression $\neg A$, where A is an atom, means, intuitively, "A is false"; the expression *not* A is interpreted as "there is no evidence that A is true." There is a clear difference between these two assertions if the program gives no information about the truth value of A.

The general form of an extended rule is

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, \tag{4}$$

where each L_i is a literal, that is, an atom possibly preceded by $\neg$.

An extended program is a set of such rules. Here is an example:

$$\begin{aligned}
&p, \\
&\neg q \leftarrow p, \\
&r \leftarrow \neg p, \\
&t \leftarrow \neg q, not\ s, \\
&u \leftarrow not\ \neg u.
\end{aligned} \tag{5}$$

Intuitively, these rules say:

> p is true[3];
> q is false if p is true;
> r is true if p is false;
> t is true if q is false and there is no evidence that s is true;
> u is true if there is no evidence that it is false.

The answers that an implementation of this language is supposed to give to the ground queries are:

$$\begin{array}{rl} p: & \textit{yes}, \\ q: & \textit{no}, \\ r: & \textit{unknown}, \\ s: & \textit{unknown}, \\ t: & \textit{yes}, \\ u: & \textit{yes}. \end{array}$$

The semantics of extended logic programs defines when a set of ground literals is an *answer set* of a program [12]. A rule with variables is treated as shorthand for the set of its ground instances. For extended programs without variables, answer sets are defined in two steps.

First, let Π be an extended program without variables that doesn't contain *not*. The *answer set* of Π is the smallest set S of ground literals such that

(i) for any rule $L_0 \leftarrow L_1, \ldots, L_m$ from Π, if $L_1, \ldots, L_m \in S$, then $L_0 \in S$;

(ii) if S contains a pair of complementary literals, then S is the set of all ground literals.

Now let Π be *any* extended program without variables. For any set S of ground literals, let Π^S be the extended program obtained from Π by deleting

(i) each rule that has an expression *not* L in its body with $L \in S$, and

(ii) all expressions of the form *not* L in the bodies of the remaining rules.

Clearly, Π^S doesn't contain *not*, so that its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π.

It is easy to check, for instance, that the program (5) has one answer set, $\{p, \neg q, t, u\}$.

The answer sets of a program can be easily characterized in terms of default logic. We will identify the rule (4) with the default

$$L_1 \wedge \ldots \wedge L_m \; : \; \overline{L_{m+1}}, \ldots, \overline{L_n} \; / \; L_0 \tag{6}$$

($\overline{L}$ stands for the literal complementary to L). Thus every extended program can be viewed as a default theory. The answer sets of a program are simply its extensions in the sense of default logic, intersected with the set of ground literals ([12], Proposition 3).

Two other approaches to the semantics of logic programs with two kinds of negation are proposed in [29] and [28]. In the context of this paper, they can be shown to lead to the same result as the answer set semantics.

4 Describing Actions by Logic Programs

Now we are ready to define the translation π from $\mathcal{A}$ into the language of extended programs.

About two different effect propositions we say that they are *similar* if they differ only by their preconditions. Our translation method is defined for any domain description that does not contain similar effect propositions. This condition prohibits, for instance, combining in the same domain such propositions as

$$\begin{array}{l} \mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Alive}\ \textbf{if}\ \mathit{Loaded}, \\ \mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Alive}\ \textbf{if}\ \mathit{VeryNervous}. \end{array}$$

(*VeryNervous* refers to the victim, of course—not to the gun.)

Let D be a domain description without similar effect propositions. The corresponding logic program πD uses variables of three sorts: *situation* variables $s, s', \ldots$, *fluent* variables $f, f', \ldots$, and *action* variables $a, a', \ldots$.[4] Its only situation constant is $S0$; its fluent constants and action constants

[4]Using a sorted language implies, first of all, that all atoms in the rules of the program are formed in accordance with the syntax of sorted predicate logic. Moreover, when we speak of an *instance* of a rule, it will be always assumed that the terms substituted for variables are of appropriate sorts.

are, respectively, the fluent names and action names of D. There are also some predicate and function symbols; the sorts of their arguments and values will be clear from their use in the rules below.

The program πD will consist of the translations of the individual propositions from D and the four standard rules:

$$\begin{array}{l} \mathit{Holds}(f, \mathit{Result}(a,s)) \leftarrow \mathit{Holds}(f,s), \mathit{not}\ \mathit{Noninertial}(f,a,s), \\ \neg \mathit{Holds}(f, \mathit{Result}(a,s)) \leftarrow \neg \mathit{Holds}(f,s), \mathit{not}\ \mathit{Noninertial}(f,a,s), \end{array} \tag{7}$$

$$\begin{array}{l} \mathit{Holds}(f,s) \leftarrow \mathit{Holds}(f, \mathit{Result}(a,s)), \mathit{not}\ \mathit{Noninertial}(f,a,s), \\ \neg \mathit{Holds}(f,s) \leftarrow \neg \mathit{Holds}(f, \mathit{Result}(a,s)), \mathit{not}\ \mathit{Noninertial}(f,a,s). \end{array} \tag{8}$$

These rules are motivated by the "commonsense law of inertia," according to which the value of a fluent after performing an action is normally the same as before. The rules (7) allow us to apply the law of inertia in reasoning "from the past to the future": the first—when a fluent is known to be true in the past, and the second—when it is known to be false. The rules (8) play the same role for reasoning "from the future to the past." The auxiliary predicate *Noninertial* is essentially an "abnormality predicate" [22].

Now we will define how π translates value propositions and effect propositions. The following notation will be useful: For any fluent name F,

$$|F| \text{ is } F, \quad |\neg F| \text{ is } F,$$

and, if t is a situation term, $\mathit{Holds}(\neg F, t)$ stands for $\neg \mathit{Holds}(F, t)$. The last convention allows us to write $\mathit{Holds}(F, t)$ even when F is a fluent name preceded by $\neg$. Furthermore, if $A_1, \ldots, A_m$ are action names, $[A_1; \ldots; A_m]$ stands for the term

$$\mathit{Result}(A_m, \mathit{Result}(A_{m-1}, \ldots, \mathit{Result}(A_1, S0) \ldots)).$$

It is clear that every situation term without variables can be represented in this form.

The translation of a value proposition (1) is

$$\mathit{Holds}(F, [A_1; \ldots; A_m]). \tag{9}$$

For instance, π(**initially** *Alive*) is

$$\mathit{Holds}(\mathit{Alive}, S0),$$

and $\pi(\neg Alive$ **after** $Shoot)$ is

$$\neg Holds(Alive, Result(Shoot, S0)).$$

The translation of an effect proposition (2) consists of $2n + 2$ rules. The first of them is

$$Holds(F, Result(A, s)) \leftarrow Holds(P_1, s), \ldots, Holds(P_n, s). \tag{10}$$

It allows us to prove that F will hold after A, if the preconditions are satisfied. The second rule is

$$Noninertial(|F|, A, s) \leftarrow not\ \overline{Holds(P_1, s)}, \ldots, not\ \overline{Holds(P_n, s)} \tag{11}$$

($\overline{Holds(P_i, s)}$ is the literal complementary to $Holds(P_i, s)$.) It disables the inertia rules (7), (8) in the cases when f can be affected by a. Without this rule, the program would be contradictory: We would prove, using a rule of the form (10), that an unloaded gun becomes loaded after the action *Load*, and also, using the second of the rules (7), that it remains unloaded!

Note the use of *not* in (11). We want to disable the inertia rules not only when the preconditions for the change in the value of F are known to hold, but whenever *there is no evidence that they do not hold*. If, for instance, we do not know whether *Loaded* currently holds, then we do not want to conclude by inertia that the value of *Alive* will remain the same after *Shoot*. We cannot draw any conclusions about the new value of *Alive*. If we replaced the body of (11) by $Holds(P_1, s), \ldots, Holds(P_n, s)$, the translation would become unsound.

Besides (10) and (11), the translation of (2) contains, for each i $(1 \leq i \leq n)$, the rules

$$Holds(P_i, s) \leftarrow \overline{Holds(F, s)}, Holds(F, Result(A, s)) \tag{12}$$

and

$$\begin{aligned} \overline{Holds(P_i, s)} \leftarrow\ & \overline{Holds(F, Result(A, s))}, \\ & Holds(P_1, s), \ldots, Holds(P_{i-1}, s), \\ & Holds(P_{i+1}, s), \ldots, Holds(P_n, s). \end{aligned} \tag{13}$$

The rules (12) justify the following form of reasoning: If the value of F has changed after performing A, then we can conclude that the preconditions were satisfied when A was performed. These rules would be unsound in the presence of similar propositions. The rules (13) allow us to conclude that a precondition was false from the fact that performing an action did not lead to the result described by an effect axiom, while all other preconditions were true.

We will illustrate the translation process by applying it to Yale Shooting (Example 2). The translation of that domain includes, in addition to (7) and (8), the following rules:

$Y1$. $\neg Holds(Loaded, S0)$.

$Y2$. $Holds(Alive, S0)$.

$Y3$. $Holds(Loaded, Result(Load, s))$.

$Y4$. $Noninertial(Loaded, Load, s)$.

$Y5$. $\neg Holds(Alive, Result(Shoot, s)) \leftarrow Holds(Loaded, s)$.

$Y6$. $Noninertial(Alive, Shoot, s) \leftarrow not\ \neg Holds(Loaded, s)$.

$Y7$. $Holds(Loaded, s) \leftarrow Holds(Alive, s), \neg Holds(Alive, Result(Shoot, s))$.

$Y8$. $\neg Holds(Loaded, s) \leftarrow Holds(Alive, Result(Shoot, s))$.

$Y9$. $\neg Holds(Loaded, Result(Shoot, s))$.

$Y10$. $Noninertial(Loaded, Shoot, s)$.

It is instructive to compare this set of rules with the formalization of Yale Shooting given by Apt and Bezem [1], who were only interested in temporal projection problems, and did not use classical negation. Instead of our four inertia rules, they have one, corresponding to the first of the rules (7). In addition, their program includes counterparts of $Y2$, $Y3$, $Y5$ and $Y6$. It does not tell us whether *Loaded* holds in the initial situation, but the negative answer to this question follows by the closed world assumption. Their rule corresponding to $Y5$ does not have $\neg$ in the head, of course; instead, the new fluent *Dead* is used. In their counterpart of $Y6$, the combination $not\ \neg$ is missing; this does not lead to any difficulties, because the closed world assumption is implicitly postulated.

5 Soundness Theorem

We say that a ground literal L is *entailed* by an extended logic program, if it belongs to all its answer sets (or, equivalently, to all its extensions in the sense of default logic). Using this notion of entailment and the entailment relation for the language $\mathcal{A}$ introduced in Section 2, we can state a result expressing the soundness of the translation π.

Soundness Theorem. *Let D be a domain description without similar effect propositions. For any value proposition P, if πD entails πP, then D entails P.*

For an inconsistent D, the statement of the soundness theorem is trivial, because such D entails every value proposition. For consistent domain descriptions, the statement of the theorem is an immediate consequence of the following lemma, which will be proved in Section 7:

Soundness Lemma. *Let D be a consistent domain description without similar effect propositions. There exists an answer set Z of πD such that, for any value proposition P, if $\pi P \in Z$ then D entails P.*

Note that the lemma asserts the possibility of selecting Z uniformly for all P; this is more than is required for the soundness theorem.

The set Z from the statement of the lemma is obviously consistent, because a consistent domain description cannot entail two complementary value propositions. Consequently, if D is consistent and does not include similar value propositions, then πD has a consistent answer set.

The converse of the soundness theorem does not hold, so that the translation π is incomplete. This following simple counterexample belongs to Thomas Woo (personal communication). Let D be the domain with one fluent name F and one action name A, characterized by two propositions:

$$F \textbf{ after } A,$$
$$A \textbf{ causes } F \textbf{ if } F.$$

It is clear that D entails **initially** F. But the translation of this proposition, $Holds(F, S0)$, is not entailed by πD. Indeed, it is easy to verify that the set of all positive ground literals other than $Holds(F, S0)$ is an answer set of πD.

6 Answer Sets and Signings

To prove the soundness lemma, we need the following definition. Let Π be a general logic program (that is, an extended program that does not contain classical negation). A *signing* for Π is any set S of ground atoms such that, for any ground instance

$$B_0 \leftarrow B_1, \ldots, B_m, \mathit{not}\ B_{m+1}, \ldots, \mathit{not}\ B_n$$

of any rule from Π, either

$$B_0, B_1, \ldots, B_m \in S,\ B_{m+1}, \ldots, B_n \notin S$$

or

$$B_0, B_1, \ldots, B_m \notin S,\ B_{m+1}, \ldots, B_n \in S.$$ [5]

For example, $\{p\}$ is a signing for the program

$$p \leftarrow \mathit{not}\ q, \quad q \leftarrow \mathit{not}\ p, \quad r \leftarrow q.$$

In this section we show that the answer sets of a general program Π which has a signing S can be characterized in terms of the fixpoints of a monotone operator. Specifically, for any set X of ground atoms, let θX be the symmetric difference of X and S:

$$\theta X = (X \setminus S) \cup (S \setminus X).$$

Obviously, θ is one to one. Moreover, it is clear that θ is an involution:

$$\begin{aligned} \theta^2 X &= \{[(X \setminus S) \cup (S \setminus X)] \setminus S\} \cup \{S \setminus [(X \setminus S) \cup (S \setminus X)]\} \\ &= (X \setminus S) \cup (S \cap X) \\ &= X. \end{aligned}$$

We will define a monotone operator ϕ such that any X is an answer set of Π if and only if θX is a fixpoint of ϕ.

Recall that, for general logic programs, the notion of an answer set (or "stable model") can be defined by means of the following construction [11]. Let Π be a general logic program, with every rule replaced by all its ground instances. The *reduct* Π^X of Π relative to a set X of ground atoms is obtained from Π by deleting

[5] This is slightly different from the original definition [18].

(i) each rule that has an expression of the form *not* B in its body with $B \in X$, and

(ii) all expressions of the form *not* B in the bodies of the remaining rules.

Clearly, Π^X is a positive program, and we can consider its "minimal model"—the smallest set of ground atoms closed under its rules. If this set coincides with X, then X is an *answer set* of Π.

This condition can be expressed by the equation $X = \alpha\Pi^X$, where α is the operator that maps any positive program to its minimal model.

Let S be a signing for Π. The operator ϕ is defined by the equation

$$\phi X = \theta\alpha\Pi^{\theta X}.$$

Lemma 1. *A set X of ground atoms is an answer set of Π iff θX is a fixpoint of ϕ.*

Proof. By the definition of ϕ, θX is a fixpoint of ϕ iff

$$\theta\alpha\Pi^{\theta^2 X} = \theta X.$$

Since θ is one-to-one and an involution, this is equivalent to

$$\alpha\Pi^X = X.$$

Note that, since θ is an involution, Lemma 1 can be also stated as follows: X is an answer set of Π iff $X = \theta Y$ for some fixpoint Y of ϕ.

Lemma 2. *The operator ϕ is monotone.*

Proof. Let Π_1 be the set of all rules from Π whose heads belong to S, and let Π_2 be the set of all remaining rules. Clearly, for any X,

$$\Pi^X = \Pi_1^X \cup \Pi_2^X.$$

Since S is a signing for Π, all atoms occurring in Π_1^X belong to S, and all atoms occurring in Π_2^X belong to the complement of S. Consequently, Π_1^X and Π_2^X are disjoint, and

$$\alpha\Pi^X = \alpha\Pi_1^X \cup \alpha\Pi_2^X.$$

Furthermore, for any expression of the form *not* B occurring in Π_1, B does not belong to S; consequently,

$$\Pi_1^X = \Pi_1^{X \setminus S}.$$

Similarly, for any expression of the form *not* B occurring in Π_2, B belongs to S, so that

$$\Pi_2^X = \Pi_2^{X \cap S}.$$

Consequently, for every X,

$$\alpha\Pi^X = \alpha\Pi_1^{X \setminus S} \cup \alpha\Pi_2^{X \cap S}.$$

In particular,

$$\alpha\Pi^{\theta X} = \alpha\Pi_1^{\theta X \setminus S} \cup \alpha\Pi_2^{\theta X \cap S}.$$

It is clear from the definition of θ that

$$\theta X \setminus S = X \setminus S,$$

$$\theta X \cap S = S \setminus X.$$

We conclude that

$$\alpha\Pi^{\theta X} = \alpha\Pi_1^{X \setminus S} \cup \alpha\Pi_2^{S \setminus X}.$$

By the choice of Π_1 and Π_2, $\alpha\Pi_1^{X \setminus S}$ is contained in S, and $\alpha\Pi_2^{S \setminus X}$ is disjoint with S. Consequently,

$$\alpha\Pi^{\theta X} \setminus S = \alpha\Pi_2^{S \setminus X},$$

$$S \setminus \alpha\Pi^{\theta X} = S \setminus \alpha\Pi_1^{X \setminus S}.$$

Hence

$$\phi X = \theta\alpha\Pi^{\theta X} = (\alpha\Pi^{\theta X} \setminus S) \cup (S \setminus \alpha\Pi^{\theta X}) = \alpha\Pi_2^{S \setminus X} \cup (S \setminus \alpha\Pi_1^{X \setminus S}).$$

Since α is monotone, and the reduct operators $X \mapsto \Pi_i^X$ are antimonotone, it follows that ϕ is monotone.

Having proved Lemmas 1 and 2, we can use properties of the fixpoints of monotone operators given by the Knaster-Tarski theorem [34] to study the answer sets of a program with a signing. The Knaster-Tarski theorem asserts, for instance, that every monotone operator has a fixpoint; this gives a new, and more direct, proof of the fact that every general program with a signing has at least one answer set.[6] Moreover, it asserts that a monotone operator has a least fixpoint, which is also its least pre-fixpoint. (A *pre-fixpoint* of ϕ is any set X such that $\phi X \subset X$.) This characterization of the least fixpoint of ϕ is used in the proof of the soundness lemma below.

7 Proof of the Soundness Lemma

The results of the previous section are not directly applicable to programs with classical negation. It is known, however, that any extended program Π can be converted into a closely related program without classical negation, as follows [12]. For each predicate P occurring in Π, select a new predicate P' of the same arity. The atom $P'(\ldots)$ is the *positive form* of the negative literal $\Pi P(\ldots)$; every positive literal is, by definition, its own positive form. The positive form of a literal L is denoted by L^+. For any set X of literals, X^+ stands for the set of the positive forms of the elements of X. For any program Π, its *positive form* is the program obtained from Π by replacing each rule (4) by

$$L_0^+ \leftarrow L_1^+, \ldots, L_m^+, not\ L_{m+1}^+, \ldots, not\ L_n^+.$$

According to Proposition 2 from [12], a consistent set X of ground literals is an answer set of Π if and only if X^+ is an answer set of the positive form of Π.

In particular, the positive form of πD has three predicate symbols: *Holds*, *Holds'* and *Noninertial*. (There is no *Noninertial'*, because the predicate *Noninertial* does not occur in πD under $\neg$.) Its rules are obtained from the rules of πD by substituting *Holds'* for $\neg$*Holds*. For instance, the inertia rules (7), (8) become

$$\begin{array}{l} Holds(f, Result(a, s)) \leftarrow Holds(f, s), not\ Noninertial(f, a, s), \\ Holds'(f, Result(a, s)) \leftarrow Holds'(f, s), not\ Noninertial(f, a, s), \end{array} \tag{14}$$

$$\begin{array}{l} Holds(f, s) \leftarrow Holds(f, Result(a, s)), not\ Noninertial(f, a, s), \\ Holds'(f, s) \leftarrow Holds'(f, Result(a, s)), not\ Noninertial(f, a, s). \end{array} \tag{15}$$

[6]The existence of answer sets for such programs, and for programs of some more general types, was established by Phan Minh Dung [5] and François Fages [8].

The rules (11) turn into

$$Noninertial(|F|, A, s) \leftarrow not\ \overline{Holds(P_1, s)}^+, \ldots, not\ \overline{Holds(P_n, s)}^+. \quad (16)$$

(The predicate symbol in the atom $\overline{Holds(P_k, s)}^+$ is either *Holds* or *Holds'*, depending on whether or not P_k includes a negation sign.)

In the rest of this section, D is a consistent domain description such that every two similar value propositions from D are disjoint, and Π stands for the positive form of πD.

Let S be the set of all ground atoms that contain the predicate symbol *Noninertial*. It is easy to see that S is a signing for Π. By θ and ϕ we denote the operators defined, for these Π and S, as in the previous section.

Recall that our goal is to find an answer set Z of πD such that, for any value proposition P, if $\pi P \in Z$, then D entails P. This set Z will be defined by the condition $Z^+ = \theta Y$, where Y is the least fixpoint of ϕ. It is easy to understand why this is a reasonable choice. Lemma 1 tells us that θY is an answer set of Π; it follows that Z is indeed an answer set of πD (provided that it is consistent). On the other hand, since Y is the *least* fixpoint of ϕ, θY includes "few" atoms beginning with *Holds* or *Holds'* (it is clear that such an atom belongs to θY iff it belongs to Y). For this reason, Z includes "few" literals with the predicate symbol *Holds*, which makes the assumption $\pi P \in Z$ in the statement of the soundness lemma particlularly strong.

For any model M of D, let $h(M)$ stand for the set of atoms of the form $(\pi P)^+$, where P is a value proposition that is true in M. It is clear that the predicate symbols in these atoms are *Holds* and *Holds'*. By $n(M)$ we denote the set of atoms of the form $Noninertial(F, A, [A_1; \ldots; A_m])$, where F is a fluent name and $A, A_1, \ldots, A_m$ are action names, such that the value propositions

$$\begin{array}{l} F\ \textbf{after}\ A_1; \ldots; A_m, \\ F\ \textbf{after}\ A_1; \ldots; A_m; A \end{array} \quad (17)$$

are either both true in M or both false in M. Finally, define

$$X_M = h(M) \cup n(M).$$

Note that $X_M \setminus S = h(M)$ and $S \setminus X_M = S \setminus n(M)$, so that

$$\theta X_M = h(M) \cup (S \setminus n(M)). \quad (18)$$

Our goal is to show that X_M is a pre-fixpoint of ϕ, that is,

$$\phi X_M \subset X_M.$$

(Lemma 5 below). To this end, we will check that X_M contains both $\phi X_M \cap S$ and $\phi X_M \setminus S$.

Lemma 3. *For any model M of D, $\phi X_M \cap S \subset X_M$.*

Proof. Assume that $B \in \phi X_M \cap S$. Then $B \in S$, which means that B has the form $\mathit{Noninertial}(F, A, [\vec{A}])$, where F is a fluent name, A is an action names, and $\vec{A}$ is a tuple $A_1; \dots; A_m$ of action names. Assume that $B \notin X_M$. Then $B \notin n(M)$, so that one of the atoms (17) is true in M, and the other false. This can be also expressed by saying that F holds in exactly one of the two states

$$M^{\vec{A}},\ \Phi(A, M^{\vec{A}}),$$

where Φ is the transition function of M. This is only possible if D includes an effect proposition describing the effect of A on F or on $\neg F$, whose preconditions hold in $M^{\vec{A}}$. Consider the rule of the type (16) corresponding to this effect proposition:

$$\mathit{Noninertial}(F, A, s) \leftarrow \mathit{not}\ \overline{\mathit{Holds}(P_1, s)}^{\,+}, \dots, \mathit{not}\ \overline{\mathit{Holds}(P_n, s)}^{\,+}.$$

The ground instance of this rule, obtained by substituting $[\vec{A}]$ for s, can be written as

$$B \leftarrow \mathit{not}\ \overline{\mathit{Holds}(P_1, [\vec{A}])}^{\,+}, \dots, \mathit{not}\ \overline{\mathit{Holds}(P_n, [\vec{A}])}^{\,+}. \tag{19}$$

Since all preconditions P_i hold in $M^{A_1;\dots;A_m}$, each of the value propositions

$$P_i\ \textbf{after}\ \vec{A}$$

is true in M. It follows that the atoms $\overline{\mathit{Holds}(P_i, [\vec{A}])}^{\,+}$ do not belong to $h(M)$. By (18), we can conclude that they do not belong to θX_M either. Consequently, the reduct $\Pi^{\theta X_M}$ includes the rule obtained by removing all expressions

$$\mathit{not}\ \overline{\mathit{Holds}(P_i, [\vec{A}])}^{\,+}$$

from (19), so that $B \in \Pi^{\theta X_M}$, and hence $B \in \alpha\Pi^{\theta X_M}$. Since B belongs also to S, it follows that

$$B \notin \theta\alpha\Pi^{\theta X_M} = \phi X_M,$$

contrary to the assumption that $B \in \phi X_M \cap S$.

Lemma 4. *For any model M of D, $\alpha\Pi^{\theta X_M} \subset h(M) \cup S$.*

Proof. It is sufficient to verify that $h(M) \cup S$ is closed under all rules of $\Pi^{\theta X_M}$. There are rules of two kinds in this program: those in which every atom belongs to S, and those in which every atom belongs to the complement of S. Consequently, we need to check that S is closed under all rules of the first kind, and $h(M)$ is closed under all rules of the second kind. The rules of the first kind are simply ground atoms beginning with *Noninertial*, so that the first claim is trivial. Let R be a rule of the second kind. It is obtained from an instance of the positive form of one of the rules of πD by deleting all expressions of the form *not* B from its body. Consider several cases, depending on the form of this rule of πD.

Case 1: R is obtained from one of the rules (7), (8). Then the positive form of this rule is one of the rules (14), (15). The ground instances of these rules have the forms

$$\begin{array}{l}
Holds(F,[\vec{A};A]) \leftarrow Holds(F,[\vec{A}]), \mathit{not}\ Noninertial(F,A,[\vec{A}]),\\
Holds'(F,[\vec{A};A]) \leftarrow Holds'(F,[\vec{A}]), \mathit{not}\ Noninertial(F,A,[\vec{A}]),\\
Holds(F,[\vec{A}]) \leftarrow Holds(F,[\vec{A};A]), \mathit{not}\ Noninertial(F,A,[\vec{A}]),\\
Holds'(F,[\vec{A}]) \leftarrow Holds'(F,[\vec{A};A]), \mathit{not}\ Noninertial(F,A,[\vec{A}]),
\end{array}$$

where $\vec{A}$ is a tuple $A_1;\dots;A_m$ of action names. Consequently, R has one of the forms

$$\begin{array}{l}
Holds(F,[\vec{A};A]) \leftarrow Holds(F,[\vec{A}]),\\
Holds'(F,[\vec{A};A]) \leftarrow Holds'(F,[\vec{A}]),\\
Holds(F,[\vec{A}]) \leftarrow Holds(F,[\vec{A};A]),\\
Holds'(F,[\vec{A}]) \leftarrow Holds'(F,[\vec{A};A]),
\end{array}$$

that is,

$$\begin{array}{l}
\pi(F\ \textbf{after}\ \vec{A};A)^+ \leftarrow \pi(F\ \textbf{after}\ \vec{A})^+,\\
\pi(\neg F\ \textbf{after}\ \vec{A};A)^+ \leftarrow \pi(\neg F\ \textbf{after}\ \vec{A})^+,\\
\pi(F\ \textbf{after}\ \vec{A})^+ \leftarrow \pi(F\ \textbf{after}\ \vec{A};A)^+,\\
\pi(\neg F\ \textbf{after}\ \vec{A})^+ \leftarrow \pi(\neg F\ \textbf{after}\ \vec{A};A)^+.
\end{array} \tag{20}$$

Moreover, $Noninertial(F,A,[\vec{A}]) \notin \theta X_M$, because otherwise the rules would not be included in the reduct $\Pi^{\theta X_M}$. Since

$$Noninertial(F,A,[\vec{A}]) \in S,$$

it follows that

$$\mathit{Noninertial}(F, A, [\vec{A}]) \in S \setminus \theta X_M = S \cap X_M = n(M).$$

By the definition of $n(M)$, this means that the the value propositions (17) are either both true in M or both false in M. It follows that if the body of one of the rules (20) belongs to $h(M)$, then so does its head.

Case 2: R is obtained from the translation of one of the value propositions P from D. Then R is $(\pi P)^+$. Since M is a model of D, P is true in M, and $(\pi P)^+ \in h(M)$.

It remains to consider the cases when R is obtained from one of the rules (10), (12) and (13), corresponding to some effect proposition P from D. (The rules obtained from (11) belong to the first kind, discussed at the beginning of the proof.) By Φ we will denote the transition function of M.

Case 3: R is obtained from (10). Then it has the form

$$\mathit{Holds}(F, [\vec{A}; A])^+ \leftarrow \mathit{Holds}(P_1, [\vec{A}])^+, \ldots, \mathit{Holds}(P_n, [\vec{A}])^+,$$

that is,

$$\pi(F \textbf{ after } \vec{A}; A)^+ \leftarrow \pi(P_1 \textbf{ after } \vec{A})^+, \ldots, \pi(P_n \textbf{ after } \vec{A})^+. \tag{21}$$

If all atoms in the body of (21) belong to $h(M)$, then all preconditions $P_1, \ldots, P_n$ hold in the state $M^{\vec{A}}$. Consequently, F holds in the state $\Phi(A, M^{\vec{A}})$, which means that the head of (21) belongs to $h(M)$.

Case 4: R is obtained from the rule (12). Assume for definiteness that F is a fluent name not preceded by $\neg$. R has the form

$$\mathit{Holds}(P_i, [\vec{A}])^+ \leftarrow \mathit{Holds}(\neg F, [\vec{A}])^+, \mathit{Holds}(F, [\vec{A}; A])^+,$$

that is,

$$\pi(P_i \textbf{ after } \vec{A})^+ \leftarrow \pi(\neg F \textbf{ after } \vec{A})^+, \pi(F \textbf{ after } \vec{A}; A)^+. \tag{22}$$

Assume that both atoms in the body of (22) belong to $h(M)$. Then F does not hold in the state $M^{\vec{A}}$ and holds in the state $\Phi(A, M^{\vec{A}})$. It follows that D includes an effect proposition P', describing the effect of A on F whose preconditions hold in $M^{\vec{A}}$. But the effect proposition P, from which R was generated, describes the effect of F on A also. Since D does not contain similar effect propositions, it follows that $P = P'$. Consequently, the preconditions of P hold in the state $M^{\vec{A}}$, and the head of (22) belongs to $h(M)$.

Case 5: R is obtained from the rule (13). Assume for definiteness that P_i and F are fluent names not preceded by $\neg$. R has the form

$$\begin{aligned} Holds(\neg P_i, [\vec{A}])^+ \leftarrow\ & Holds(\neg F, [\vec{A}; A])^+, \\ & Holds(P_1, [\vec{A}])^+, \ldots, Holds(P_{i-1}, [\vec{A}])^+, \\ & Holds(P_{i+1}, [\vec{A}])^+, \ldots, Holds(P_n, [\vec{A}])^+, \end{aligned}$$

that is,

$$\begin{aligned} \pi(\neg P_i \textbf{ after } \vec{A})^+ \leftarrow\ & \pi(\neg F \textbf{ after } \vec{A}; A)^+, \\ & \pi(P_1 \textbf{ after } \vec{A})^+, \ldots, \pi(P_{i-1} \textbf{ after } \vec{A})^+, \\ & \pi(P_{i+1} \textbf{ after } \vec{A})^+, \ldots, \pi(P_n \textbf{ after } \vec{A})^+. \end{aligned} \tag{23}$$

Assume that all atoms in the body of (23) belong to $h(M)$. Then F does not holds in the state $\Phi(A, M^{\vec{A}})$. This is only possible when at least one of the preconditions $P_1,\ldots,P_n$ does not hold in the state $M^{\vec{A}}$. But all preconditions other than P_i hold in this state; consequently, P_i does not hold, which means that the head of (23) belongs to $h(M)$.

Lemma 5. *For any model M of D, $\phi X_M \subset X_M$.*

Proof. By the definitions of ϕ and θ and Lemma 4,

$$\phi X_M \setminus S = \theta\alpha\Pi^{\theta X_M} \setminus S = \alpha\Pi^{\theta X_M} \setminus S \subset (h(M) \cup S) \setminus S \subset h(M) \subset X_M.$$

From this inclusion and Lemma 3,

$$\phi X_M = (\phi X_M \cap S) \cup (\phi X_M \setminus S) \subset X_M.$$

Lemma 6. *Let Y be the least fixpoint of ϕ. For any value proposition P, if $(\pi P)^+ \in \theta Y$, then D entails P.*

Proof. Assume that $(\pi P)^+ \in \theta Y$, and take any model M of D. By the Knaster-Tarski theorem, Y is the least pre-fixpoint of ϕ; by Lemma 5, X_M is a pre-fixpoint of ϕ. Consequently, $Y \subset X_M$. By the choice of S, $(\pi P)^+ \notin S$. Consequently,

$$(\pi P)^+ \in \theta Y \setminus S = [(Y \setminus S) \cup (S \setminus Y)] \setminus S = Y \setminus S \subset Y \subset X_M = h(M) \cup n(M).$$

Since the predicate symbol in $(\pi P)^+$ is *Holds* or *Holds'*, it follows that $(\pi P)^+ \in h(M)$, so that P is true in M.

Now we are ready to prove the soundness lemma. Assume that D is consistent. Consider the set Z of literals such that $Z^+ = \theta Y$, where Y is the least fixpoint of ϕ. By Lemma 1, Z^+ is an answer set of Π. Case 1: Z is consistent. Since Z^+ is an answer set of the positive form of πD, we can conclude that Z is an answer set of πD. If πD entails πP, then $\pi P \in Z$, and consequently $(\pi P)^+ \in Z^+ = \theta Y$. By Lemma 6, it follows that D entails P. Case 2: Z is inconsistent. This means that Z contains a pair of complementary literals L, $\overline{L}$. Since Π does not contain *Noninertial'*, its answer set Z^+ does not contain *Noninertial'* either, so that Z does not contain $\neg$*Noninertial*. Consequently, the predicate symbol in L and $\overline{L}$ has to be *Holds*. Then these literals can be obtained by applying π to two complementary value propositions. By the choice of X, these value propositions are both entailed by D. This is impossible, in view of the consistency of D.

8 Conclusions and Future Work

This paper is the first step in the development of high-level languages designed specifically for representing actions. The syntax and semantics of $\mathcal{A}$ precisely describe the class of action domains under consideration and the intended ontology of action. The representation of a particular domain in $\mathcal{A}$ can be viewed as a high-level specification for the task of formalizing this domain in logic programming or another logic-based formalism. The soundness and completeness of each formalization become precisely stated mathematical questions. The possibilities and limitations of different representation methods can be compared in a precise fashion. For instance, in [15] this approach is used to prove the equivalence of the methods for formalizing actions proposed earlier by Pednault [26], Reiter [31] and Baker [2] for the domains representable in $\mathcal{A}$.

On the other hand, this paper is one of the first experiments (along with [17], [27], [10]) on using extended logic programs for representing knowledge. Not much is known yet about mathematical properties of extended programs. For this reason, in this initial experiment, the source language $\mathcal{A}$ was deliberately made quite simple, and we did not try to make the translation complete. As we have seen, even the soundness theorem limited to this class of domains turns out to be nontrivial.

The next step will be to make the translation complete and applicable to domain descriptions containing similar propositions. It appears that both goals can be achieved by using the more expressive language of *disjunctive* programs [12] as the target language for the translation. The head of a disjunctive rule is a list of literals separated by occurrences of the "epistemic disjunction" symbol |. For example, each of the rules (12) can be replaced by the more powerful disjunctive rule

$$Holds(F, s) \mid Holds(P_i, s) \leftarrow Holds(F, Result(A, s)).$$

This will apparently eliminate the cases of incompleteness similar to the counterexample from the end of Section 5. Similarly, all n rules (13) can be replaced by the more intuitive disjunctive rule

$$\overline{Holds(P_1, s)} \mid \ldots \mid \overline{Holds(P_n, s)} \leftarrow \overline{Holds(F, Result(A, s))}.$$

Another useful extension of this work made possible by using disjunctive rules has to do with disjunctive information about the initial situation. In the dialect of $\mathcal{A}$ that allows us to represent such information, a value proposition may include a disjunction of fluent expressions (or, more generally, an arbitrary propositional combination of fluent names) in place of a single fluent expression. For instance, in a "Russian roulette" version of the shooting example, we have two guns, described by two fluents, *Loaded1* and *Loaded2*, and the initial condition can be

$$\textbf{initially } Loaded1 \vee Loaded2. \tag{24}$$

In the corresponding logic program, (24) will be represented by the disjunctive rule

$$Holds(Loaded1, S0) \mid Holds(Loaded2, S0).$$

Extending the semantics of $\mathcal{A}$ to this dialect is straightforward. However, generalizing the soundness theorem to disjunctive value propositions requires further work on the mathematics of disjunctive programs.

The shooting domain with several guns is one of the cases when "first-order" notation would be more natural than the "propositional" notation of $\mathcal{A}$. We can write

$$\textbf{initially } Loaded(Gun1) \vee Loaded(Gun2)$$

instead of (24), and express the main property of shooting by the schema

$$Shoot(x)\ \mathbf{causes}\ \neg Alive\ \mathbf{if}\ Loaded(x), \tag{25}$$

where x is a metavariable for the expressions *Gun1*, *Gun2*. Thus (25) is viewed as shorthand for the collection of its ground instances, which are propositions in $\mathcal{A}$; no extension of the semantics of $\mathcal{A}$ is needed.

Proposition (25) can be translated into logic programming directly, by rules like

$$\neg Holds(Alive, Result(Shoot(x), s)) \leftarrow Holds(Loaded(x), s).$$

Here x is again a "variable for guns." Indeed, the ground instances of this rule are identical to the ground instances of the rules

$$\begin{array}{l} \neg Holds(Alive, Result(Shoot(Gun1), s)) \leftarrow Holds(Loaded(Gun1), s), \\ \neg Holds(Alive, Result(Shoot(Gun2), s)) \leftarrow Holds(Loaded(Gun2), s), \end{array}$$

corresponding to the two instances of (25).

We are working on developing extensions of $\mathcal{A}$ capable of expressing richer ontologies of actions.

The most striking limitation of $\mathcal{A}$ is its inability to express *domain constraints*. The fluents represented in $\mathcal{A}$ are presumed to be independent, in the sense that the semantics of $\mathcal{A}$ treats any assignment of truth values to the fluent constants as a valid state.

Syntactically, constraints will be expressed by propositions of the form

$$\mathbf{always}\ \ < formula > .$$

For instance, we can express that an object cannot occupy two locations at once by the proposition

$$\mathbf{always}\ \neg(At(x, l_1) \wedge At(x, l_2))$$

for all x, l_1, l_2 such that $l_1 \neq l_2$. Semantically, including constraints will require that a state be defined as a truth assignment to the fluent constants that makes all constraint formulas true. Another necessary change in the semantics is due to the fact that, in the presence of constraints, an action may have indirect effects. For instance, consider the action of moving x from

l_1 to l_2. If the only explicitly given effect of this action is that it makes $At(x, l_2)$ true, we should be able to conclude that it also makes $At(x, l_1)$ false (because otherwise a constraint would be violated).

We plan to design and investigate dialects of $\mathcal{A}$ in which nondeterministic actions can be described. In fact, nondeterminism is closely related to the idea of ramifications, since the indirect effects of an action can be nondeterministic. Almost nothing is currently known about the frame problem in the presence of nondeterminism. One way to include nondeterminism is to allow effect propositions to contain disjunctions, for instance:

$$\mathit{TossCoin}\ \mathbf{causes}\ \mathit{Heads} \lor \mathit{Tails}.$$

Semantically, in either case, nondeterministic transition functions will be used. In the corresponding logic program, the effect of *TossCoin* will be expressed by a disjunctive rule.

In [3], the extension of $\mathcal{A}$ is introduced in which one can describe the concurrent execution of actions. In this extension, performing several actions concurrently can be represented by using a set of action names instead of a single action name in a proposition, for instance:

$$\mathit{Alive}\ \mathbf{after}\ \{\mathit{Wait}\}\ \{\mathit{Shoot}(\mathit{Gun1}), \mathit{Load}(\mathit{Gun2})\},$$

and the semantics of $\mathcal{A}$ is generalized accordingly. The translation to logic programming presented here is extended to this "concurrent $\mathcal{A}$" in the spirit of [13].

The inconsistency of the Stolen Car domain (Example 4) illustrates the fact that $\mathcal{A}$ cannot be used for representing "causal anomalies," or "miracles" [21]. We plan to address this issue in further work, too. Our preferred approach to causal anomalies is to view them as evidence of unknown events that occur concurrently with the given actions and contribute to the properties of the new situation.

One other dialect of $\mathcal{A}$ is described in [19]. It has symbols for temporal intervals over which actions may occur.

A referee has pointed out to us that there is a simple and elegant translation from $\mathcal{A}$ into a form of abductive logic programming with integrity constraints, which, inlike the method of [33], handles all forms of temporal reasoning in a uniform way. It would be interesting to extend this translation to more expressive dialects of $\mathcal{A}$ also.

Acknowledgements

We would like to thank G. N. Kartha and Norman McCain for comments on a draft of this paper, Kenneth Kunen for directing us to his paper on signings, Thomas Woo for the counterexample reproduced in Section 5, and the referees for valuable criticisms. This research was supported in part by NSF grants CDA-9015006, IRI-9101078 and IRI-9103112.

References

[1] Krzysztof Apt and Marc Bezem. Acyclic programs. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 617–633, 1990.

[2] Andrew Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[3] Chitta Baral and Michael Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of IJCAI-93*, pages 866–871, 1993.

[4] Weidong Chen and David S. Warren. A goal-oriented approach to computing well-founded semantics. In Krzysztof Apt, editor, *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, pages 589–603, 1992.

[5] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[6] Kave Eshghi and Robert Kowalski. Abduction compared with negation as failure. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. of the Sixth Int'l Conf.*, pages 234–255, 1989.

[7] Chris Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proc. of the Second Int'l Symp. on Artificial Intelligence*, 1989.

[8] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[9] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[10] Michael Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 1993. To appear.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[12] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[13] Michael Gelfond, Vladimir Lifschitz, and Arkady Rabinov. What are the limitations of the situation calculus? In Robert Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 167–179. Kluwer Academic, Dordrecht, 1991.

[14] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[15] G. Neelakantan Kartha. Soundness and completeness theorems for three formalizations of action. In *Proc. of IJCAI-93*, pages 724–729, 1993.

[16] Henry Kautz. The logic of persistence. In *Proc. of AAAI-86*, pages 401–405, 1986.

[17] Robert Kowalski and Fariba Sadri. Logic programs with exceptions. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 598–613, 1990.

[18] Kenneth Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[19] Vladimir Lifschitz. A language for describing actions. In *Working Papers of the Second Symposium on Logical Formalizations of Commonsense Reasoning*, 1993.

[20] Vladimir Lifschitz, Norman McCain, and Hudson Turner. Automated reasoning about action: A logic programming approach. In Dale Miller, editor, *Proc. of ILPS-93*, page 641, 1993.

[21] Vladimir Lifschitz and Arkady Rabinov. Miracles in formal theories of actions. *Artificial Intelligence*, 38(2):225–237, 1989.

[22] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986. Reproduced in [23].

[23] John McCarthy. *Formalizing common sense: papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[24] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969. Reproduced in [23].

[25] Paul Morris. The anomalous extension problem in default reasoning. *Artificial Intelligence*, 35(3):383–399, 1988.

[26] Edwin Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[27] Luis Pereira, Joaquim Aparicio, and Jose Alferes. Non-monotonic reasoning with well-founded semantics. In *Proc. of the Eight International Logic Programming Conference*, pages 475–489, 1992.

[28] Luis Pereira and Jose lferes. Well-founded semantics for logic programs with explicit negation. In *Proc. of the Tenth European Conf. on Artificial Intelligence*, pages 102–106, 1992.

[29] Teodor Przymusinski. Extended stable semantics for normal and disjunctive programs. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 459–477, 1990.

[30] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[31] Raymond Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[32] Lenhart Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

[33] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of IJCAI-89*, pages 1055–1060, 1989.

[34] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.