# Action Languages, Answer Sets and Planning

Vladimir Lifschitz
University of Texas at Austin

**Abstract**

This is a discussion of some of the achievements and challenges related to representing actions and the design of planners from the perspective of logic programming. We talk about recent work on action languages and translating them into logic programming, on representing possible histories of an action domain by answer sets, on efficient implementations of the answer set semantics and their use for generating plans, and on causal logic and its relation to planning algorithms. Recent progress in these areas may lead to the creation of planners which are based on the ideas of logic programming and combine the use of expressive action description languages with efficient computational procedures.

## 1 Introduction

This is a discussion of some of the achievements and challenges related to representing actions and the design of planners from the perspective of logic programming.

Describing the effects of actions and generating plans have been at the center of Artificial Intelligence research since the late fifties. In his classical paper *Programs with Common Sense*, John McCarthy [1959] discusses the problem of planning his way to the airport, in which an initial state, described by the formula $at(I, desk)$, needs to be turned into a goal state that satisfies $at(I, airport)$. The goal can be achieved by executing first the action $go(desk, car, walking)$, and then $go(home, airport, driving)$. The effect of $go$ is described by the axiom

$$did(go(x, y, z)) \supset at(I, y). \tag{1}$$

Other axioms tell us under what conditions it is possible to walk or drive from $x$ to $y$; the possibility of executing an action $a$ is denoted by $can(a)$.

More recently, work on representing changes caused by actions has been extended to causal dependencies that do not involve actions, including the cases when the cause and the effect are not even separated by a time interval—they obtain in the same state. If I have a watch on my wrist then $at(I, airport)$ is a cause for $at(watch, airport)$. There is no action or passage of time involved in this causal dependency; it is "static," unlike the "dynamic" dependency between going from $x$ to $y$ and being at $y$. Combinations of dynamic and static causal dependencies is what allows actions to have indirect effects, or "ramifications." When I go to the airport, $at(I, airport)$ is the direct effect of this action, and $at(watch, airport)$ is one of its many indirect effects.

Once an action domain is described by a formal theory, it may be possible to use a theorem prover for solving planning problems in this domain. The influential paper by Cordell Green [1969] entitled *Application of Theorem Proving to Problem Solving* shows, for instance, how a resolution proof procedure can be used to solve the problem of getting a bunch of bananas hanging from the ceiling just beyond a monkey's reach. The answer literal method allows us to extract the value of the situation variable $s$ from the proof of the formula

$$\exists s\, HAS(monkey, bananas, s)$$

found by this procedure. The value, which equals

$$\begin{array}{l} reach(monkey, bananas, \\ \quad climb(monkey, box, \\ \qquad move(monkey, box, under\text{-}bananas, s_0))), \end{array}$$

encodes a solution to the planning problem. Finding a plan by extracting a term from an existence proof is known as *deductive planning*.

The answer literal method was a precursor of logic programming, and it is not surprising that logic programming has always played an important role in research on describing the effects of actions and on planning.

One other reason why the study of causal dependencies is related to logic programming is that causation is noncontrapositive, just like rules in logic programs. We can say, for instance, that being underpaid usually causes a person to look for another job. But it is not true that the lack of interest in other jobs is usually a cause for being paid well. This is similar to the difference between the rules

$$p \leftarrow not\ q$$

and

$$q \leftarrow not\ p.$$

This formal similarity makes the if operator used in logic programming a better tool for formalizing causality than material implication with its contraposition property

$$\neg p \supset q \equiv \neg q \supset p.$$

Over the years, the early work referenced above has inspired many research projects and scores of papers and monographs. This note is not an attempt to survey all important ideas developed in this research. We concentrate here on a small fraction of these ideas, on just a few threads, and try to show how they interact with each other and what can be expected from them in the future.

## 2 Incomplete Information

Some examples of reasoning about actions involve the execution of an action in a state that is not completely described. Consider, for instance, three boxes $A$, $B$ and $C$. Currently $A$ is empty, but whether or not $B$ and $C$ are empty we are not told. Imagine now that an object is placed in box $B$. In the new state, $A$ is empty and $B$ is not; about $C$ we still do not know.

In classical logic, the incompleteness of information corresponds to the use of incomplete theories. When the example above is formalized as a classical theory, the set of theorems of the theory might include formulas like

$$\mathit{Empty}(A,0),\ \mathit{Empty}(A,1),\ \neg\mathit{Empty}(B,1) \tag{2}$$

(the second argument of *Empty* represents time, and we assume here that the action in question is performed between the times 0 and 1); the formulas

$$\mathit{Empty}(B,0),\ \mathit{Empty}(C,0),\ \mathit{Empty}(C,1) \tag{3}$$

would be undecidable in the theory— neither these formulas nor their negations would be among the theorems. For traditional logic programming, on the other hand, representing incomplete information is a little bit of a problem. Given a ground atom like $\mathit{Empty}(C,1)$ as a query, a Prolog system can only give the answers *yes* and *no*; it has no way of telling us that the answer is unknown (other than by not terminating). In this sense, Prolog programs are similar to *complete* theories.

The issue of representing actions by logic programs in the absence of complete information was addressed in [Gelfond and Lifschitz, 1993] using "extended" logic programs introduced in [Gelfond and Lifschitz, 1991]. In addition to the negation as failure operator *not*, an extended program can use the second negation symbol $\neg$, taken from the vocabulary of classical logic. The head of an extended rule is not necessarily an atom; it can be also a negative literal $\neg A$. (In the theory of extended logic programs, the term "literal" is reserved for an atom possibly preceded by *classical* negation, not negation as failure.) In the body of a rule, both kinds of negation are allowed. For instance,

$$\neg p \leftarrow not\ p \tag{4}$$

and

$$p \leftarrow not\ \neg p \tag{5}$$

are extended rules. Rule (4), when included in a logic program $\Pi$, expresses, intuitively, that $p$ is false if $\Pi$ provides no evidence that $p$ is true. It formalizes the "closed world assumption" for $p$. Rule (5) has the opposite meaning: $p$ is true if $\Pi$ provides no evidence that $p$ is false.

The set of consequences of an extended program is a set of literals. This set may be incomplete—it may include, for instance, literals (2), but neither atoms (3) nor their negations.

## 3 Action Language $\mathcal{A}$

Many authors who write about representing action domains explain their ideas by showing how they apply to specific examples, such as the Yale Shooting scenario from [Hanks and McDermott, 1987] and its enhancements, or various versions of the blocks world. The method for representing actions in logic programming proposed in [Gelfond and Lifschitz, 1993] is described there in a more systematic way: by defining a translation from an "action language" $\mathcal{A}$ into the language of extended logic programs.

Action languages are formal models of parts of the natural language that are used for talking about the effects of actions. The main elements of the language $\mathcal{A}$ are *effect propositions* which we will write in the form

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

Here $A$ is the name of an action, $L$ is a literal, and $F$ is a propositional formula. The atoms occurring in $L$ and $F$ represent "propositional fluents," that is, conditions that may be true or false depending on the state of the system. The formula $F$ is the precondition which guarantees that $L$ would hold after executing $A$ in the current state. To express that the execution of $A$ causes $L$ under all circumstances, we take the logical constant $\top$ ("true") as the precondition. For instance, the counterpart of (1) in the language $\mathcal{A}$ is

$$go(x,y,z)\ \mathbf{causes}\ at(I,y)\ \mathbf{after}\ \top\ .$$

The effect of the action *Shoot* on the fluent *Alive* from the Yale Shooting story can be described in $\mathcal{A}$ by the effect proposition

$$Shoot\ \mathbf{causes}\ \neg Alive\ \mathbf{after}\ Loaded\ .$$

(Shooting is not supposed to affect the value of the fluent *Alive* unless the gun is loaded prior to the execution of the action.)

According to the semantics of $\mathcal{A}$, a set of value propositions defines a *transition diagram*—a directed graph whose vertices correspond to states (functions from atoms to truth values) and whose arcs are labeled by actions. For instance, the set consisting of one proposition

$$A\ \mathbf{causes}\ P\ \mathbf{after}\ Q \tag{6}$$

represents the transition diagram shown in Figure 1.

The vertical edge in the right part of the diagram shows that the action $A$, executed in the state in which $P$ is false and $Q$ is true, would make $P$ true, in accordance with the intuitive meaning of (6).

Note, on the other hand, that the two edges in the left part of the diagram are loops. According to the semantics of the language, the values of the fluents are not affected when $A$ is executed in a state in which the precondition $Q$ is not satisfied. In this sense, the semantics of $\mathcal{A}$ incorporates a solution to (a simple special case of) the frame problem.

The main theorem of [Gelfond and Lifschitz, 1993] shows that the translation from $\mathcal{A}$ into the language of logic programming proposed in that paper is sound with respect to the semantics of $\mathcal{A}$.

## 4 Answer Sets and Histories

The set of consequences of an extended logic program is defined as the intersection of all "answer sets" (or "stable models") of the program. For

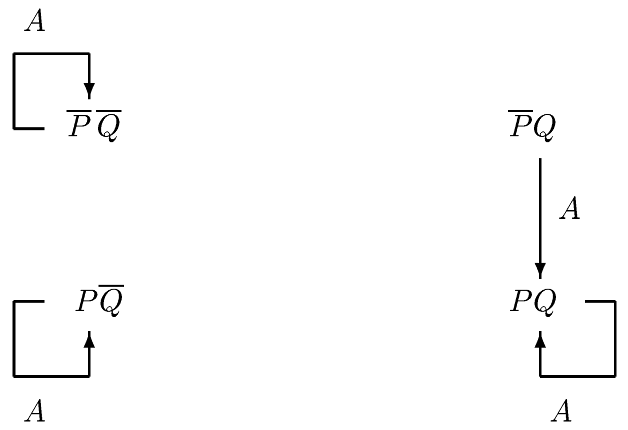


Figure 1: Transition diagram for proposition (6)

instance, the program

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ r \leftarrow p \\ r \leftarrow q \end{array} \tag{7}$$

has two answer sets: $\{p, r\}$ and $\{q, r\}$; the only consequence of this program is $r$.

A program obtained by the translation process from [Gelfond and Lifschitz, 1993] may have many answer sets, and the soundness theorem established in that paper makes no claims about individual answer sets for such a program. It only asserts that a literal which is a *consequence* of the program—that is to say, which belongs to *all* its answer sets—is true in the corresponding action domain.

Alternative translations from action languages to logic programming proposed in [Denecker and De Schreye, 1993], [Dung, 1993] and [Turner, 1997] show that there may exist a close relationship between possible histories of an action domain (paths in its transition diagram) on the one hand, and individual answer sets for a logic program on the other. This idea has led Subrahmanian and Zaniolo [1995] to the invention of a planning method which is based on logic programming but is different from deductive planning mentioned above. To find a plan that leads from a given initial condition to a given goal means to find a history corresponding to the execution of the plan, and this can be accomplished by finding an answer set for a corresponding logic program.

This method is similar to what is known as *satisfiability planning* [Kautz and Selman, 1992]—treating a planning problem as the problem of finding an interpretation satisfying a set of propositional formulas. Satisfiability planning has been successfully applied to some of the planning problems that are known to lead to serious computational difficulties in traditional planners [Kautz and Selman, 1996]. The Subrahmanian—Zaniolo proposal is different in that they suggest using propositional logic programs under the answer set semantics instead of propositional theories.

The problem of finding an answer set for a propositional logic program and the problem of finding a satisfying interpretation for a set of propositional formulas are in the same complexity class: they are both NP-hard, just like the planning problem itself (assuming that the length of the plan is restricted by a polynomial). Using logic programs is an attractive idea because encoding action domains in logic programming may be easier than encoding them in classical propositional logic.

## 5 Computing Answer Sets

To implement the planning method from [Subrahmanian and Zaniolo, 1995], we need an efficient system for computing answer sets. One such system, called SMODELS, is described in [Niemelä and Simons, 1996]. The current version of the system does not handle programs with classical negation, so that any answer set it computes is a set of atoms.

Given, for instance, the input file

```
p :- not q.
q :- not p.
r :- p.
r :- q.

compute all
```

as input, SMODELS will generate both answer sets for program (7). In the last line of the file, `all` can be replaced by a positive integer that shows how many answer sets we wish the system to compute. The user can also specify that the only answer sets to be generated are those that include, or do not include, a given atoms or atoms. For instance, the directive

```
compute 1 {not p}
```

instructs the system to generate an answer set that does not include $p$.

As the designers of SMODELS indicate, their system is different from familiar logic programming systems in that it *does not perform query evaluation*. It treats a logic program as a constraint on a set of atoms, and generates the sets that satisfy this constraint.

When SMODELS is used to solve propositional satisfiability problems encoded in the language of logic programming, its performance is comparable to the performance of efficient propositional solvers [Simons, 1997].

The availability of this system made it possible for Dimopoulos, Nebel and Koehler [1997] to conduct planning experiments using a method similar to the one proposed in [Subrahmanian and Zaniolo, 1995]. They report that, for careful and compact encodings of planning problems, the performance of their method is comparable to that of efficient general-purpose planners.

One other system that can apparently be used for planning in a similar way is DLV, the implementation of disjunctive logic programming described in [Eiter *et al.*, 1998]. The home pages of SMODELS and DLV can be accessed through the Library of Logic Programming Systems and Test Cases (`http://www.uni-koblenz.de/ag-ki/LP/lp_systems.html`). Yet another possibility is to use the implementation of default logic developed at the University of Kentucky (`http://www.cs.engr.uky.edu/~lpnmr/DeReS.html`).

## 6 Causal Reasoning

The methods for encoding action domains in logic programming proposed in [Subrahmanian and Zaniolo, 1995] and [Dimopoulos *et al.*, 1997] are not applicable in the presence of static causal dependencies. Consequently, as discussed in the introduction, they cannot be used when plans may include actions with indirect effects. To see how this limitation can be lifted, we need to consider some of the recent work on causal reasoning.

The theory we concentrate on in this section is proposed in [McCain and Turner, 1997] and generalized in [Lifschitz, 1997]. Some of its roots are found in [Geffner, 1990] and [Lin, 1995].

In this theory, we distinguish between claiming that a proposition is true and making the stronger assertion that there is a *cause* for it to be true. In the example from Section 2, for instance, about the formula $Empty(A, 0)$ we can only say that it is true, but about $\neg Empty(B, 1)$ we can also say that it is caused: its cause is that an object is placed in box $B$ between times

0 and 1. The formula $Empty(A, 1)$ holds "by inertia," and here two views are possible: we can say that there is no cause for it (Lin's position) or that inertia is the cause (as McCain and Turner would say).

A *causal rule* is an expression of the form

$$F \leftarrow G \tag{8}$$

where $F$ and $G$ are formulas of classical logic. Intuitively, rule (8) expresses that *$F$ is caused if $G$ is true*. The dynamic causal dependency expressed in the introduction by formula (1) would be written in this language as

$$at(I, y, t+1) \leftarrow go(x, y, z, t)$$

(there is a cause for me to be at location $y$ at time $t+1$ if I performed the action $go(x, y, z)$ during the unit time interval beginning at time $t$). Static causal dependencies can be represented by causal rules as well: the rule

$$at(watch, y, t) \leftarrow at(I, y, t)$$

says that whenever I am at a location $y$, there is a cause for my watch to be at $y$ also.

Note that causal rules of the form

$$p \leftarrow p \tag{9}$$

are not trivial (although a rule of this form found in a logic program can be dropped without changing the meaning of the program, according to the answer set semantics). Causal rule (9) says that if $p$ holds then there is a cause for this. Its role is similar to the role of rule (4), expressing the inverted closed world assumption, in an extended logic program.

Even though causal theories and logic programs have somewhat different properties, there exists a close relationship between them, and this relationship played an important role in the development of causal logic. The computational method for causal theories proposed in [McCain and Turner, 1997], called *literal completion*, is an elaboration of the completion method from [Clark, 1978] proposed as a semantics for negation as failure. Theorems relating causal theories to logic programs can be found in [Geffner, 1990] and [Turner, 1998].

The *causal calculator* is a system, written at the University of Texas, that can perform reasoning about actions and planning for domains described by causal theories. It uses the process of literal completion to reduce these tasks to propositional reasoning problems, which are submitted to a propositional solver. The causal calculator is written in Prolog. Some information on its perfromance can be found in [McCain and Turner, 1998].

## 7 Action Language $\mathcal{C}$

In [Giunchiglia and Lifschitz, 1998], ideas of causal logic are applied to the design of an action language that is considerably more expressive than $\mathcal{A}$. Among other things, the new language, called $\mathcal{C}$, allows us to represent static causal dependencies and to talk about the concurrent execution of actions.

In place of effect propositions of the language $\mathcal{A}$ (Section 3), $\mathcal{C}$ has proposition of two kinds: *static laws* of the form

$$\textbf{caused } F \textbf{ if } G \tag{10}$$

and *dynamic laws* of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H. \tag{11}$$

Here $F$ and $G$ are formulas whose atomic components represent fluents, as in the language $\mathcal{A}$. The formula $H$ is of a more general kind: in addition to fluent symbols, it is allowed to contain the names of actions. Syntactically, action names are treated as atomic formulas; an assignment of truth values to action names represents the composite action which is executed by performing concurrently all elementary actions whose names are assigned the value *true*. If, for instance, the description of an action domain in $\mathcal{C}$ uses the action names $A_1$ and $A_2$ then it tells us about the effects of 4 composite actions: $A_1\overline{A_2}$ (performing $A_1$ only), $\overline{A_1}A_2$ (performing $A_2$ only), $A_1A_2$ (performing both actions concurrently) and $\overline{A_1}\,\overline{A_2}$ (doing nothing).

Direct effects of actions are described in $\mathcal{C}$ by dynamic laws (11) in which $G$ is the logical constant $\top$. We will write such laws as

$$\textbf{caused } F \textbf{ after } H. \tag{12}$$

For instance, the counterpart of (6) in $\mathcal{C}$ is

$$\textbf{caused } P \textbf{ after } Q \wedge A. \tag{13}$$

The language $\mathcal{C}$ differs from $\mathcal{A}$ in that the idea of inertia is not incorporated into its semantics. If we want to assume that a certain fluent tends to keep the value it had before then an appropriate dynamic law needs to be postulated along with the laws describing the effects of actions. The expression

$$\textbf{inertial } F$$

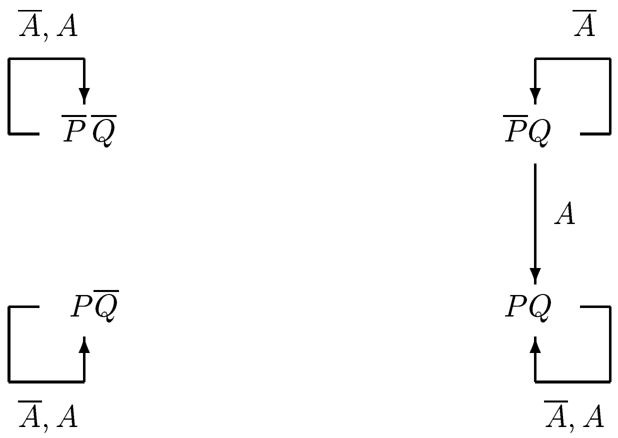


Figure 2: Transition diagram for propositions (13), (14)

stands for

$$\textbf{caused } F \textbf{ if } F \textbf{ after } F$$

(if $F$ was true before and is true now then there is a cause for this). This elegant solution to the frame problem is due to McCain and Turner [1997], who expressed it in the language of causal logic. Its advantage, in comparison with turning inertia into a "built-in" feature of the language, is that it gives the user freedom to decide which inertia assumptions are appropriate in each particular case.

For example, the formalization of the example from Section 3 in $\mathcal{C}$ includes, besides dynamic law (13), the inertia postulates

$$\begin{array}{l} \textbf{inertial } P \\ \textbf{inertial } \neg P \\ \textbf{inertial } Q \\ \textbf{inertial } \neg Q \end{array} \tag{14}$$

which we can abbreviate as

$$\textbf{inertial } P, \neg P, Q, \neg Q \ .$$

According to the semantics of $\mathcal{C}$, propositions (13) and (14) represent the transition diagram shown in Figure 2.

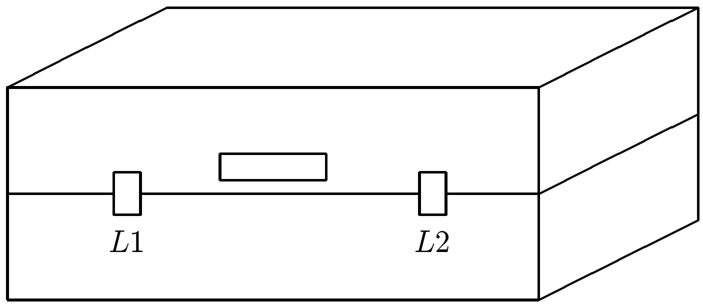


Figure 3: Fangzhen Lin's suitcase

As an example of the use of static laws in $\mathcal{C}$, consider a suitcase with two latches (Figure 3) which is spring-loaded: it is open whenever both latches are up [Lin, 1995]. The suitcase is described in $\mathcal{C}$ by the propositions

$$\begin{array}{l}
\textbf{caused } \mathit{Up}(l) \textbf{ after } \neg \mathit{Up}(l) \wedge \mathit{Toggle}(l) \\
\textbf{caused } \neg \mathit{Up}(l) \textbf{ after } \mathit{Up}(l) \wedge \mathit{Toggle}(l) \\
\textbf{caused } \mathit{Open} \textbf{ if } \mathit{Up}(L1) \wedge \mathit{Up}(L2) \\
\textbf{inertial } \mathit{Up}(l), \neg \mathit{Up}(l) \\
\textbf{inertial } \mathit{Open}, \neg \mathit{Open}
\end{array} \tag{15}$$

($l \in \{L1, L2\}$). The third line is a static law.

According to the semantics of $\mathcal{C}$, assigning the value *true* to *Open* and the value *false* to both $\mathit{Up}(L1)$ and $\mathit{Up}(L2)$ does not give a possible state of this system, so that the transition diagram for (15) has only 7 vertices. Performing any action that makes $\mathit{Up}(L1) \wedge \mathit{Up}(L2)$ true has an indirect effect—it makes *Open* true also.

## 8 From $\mathcal{C}$ to Logic Programming

In this section, we restrict attention to static laws (10) and dynamic laws (11) in which formula $F$ is a literal and each of the formulas $G$, $H$ is a conjunction of literals (or $\top$, which will be identified with the empty conjunction). All propositions from the two examples above clearly satisfy this condition.

The problem of computing the transition diagram for a set of static and dynamic laws of this kind can be reduced to generating all answer sets for a logic program.

This logic program includes, besides the fluent names and action names, a copy $P'$ of every fluent name $P$. The atoms $P$, $P'$ represent the values of the fluent at the two vertices connected by an edge of the transition diagram.

The process of translating a set of static and dynamic laws into a logic program consists of three steps.

1. A static law

$$\textbf{caused } L_0 \textbf{ if } L_1 \wedge \cdots \wedge L_m$$

is turned into the pair of rules

$$\begin{array}{l} L_0 \leftarrow L_1, \ldots, L_m \ , \\ L'_0 \leftarrow \mathit{not}\ \overline{L'_1}, \ldots, \mathit{not}\ \overline{L'_m} \ . \end{array}$$

These rules express in the language of logic programming that the static law applies to both vertices connected by the edge. Note that each member $\mathit{not}\ \overline{L'_i}$ of the body of the second rule is obtained from the literal $L'_i$ by putting the negation as failure operator in front of the complementary literal. The relation between $L'_i$ and $\mathit{not}\ \overline{L'_i}$ is similar to the relation between the bodies of causal rule (9) and of logic programming rule (5) discussed in Section 6.

2. A dynamic law

$$\textbf{caused } L_0 \textbf{ if } L_1 \wedge \cdots \wedge L_m \textbf{ after } L_{m+1} \wedge \cdots \wedge L_n$$

is turned into the rule

$$L'_0 \leftarrow \mathit{not}\ \overline{L'_1}, \ldots, \mathit{not}\ \overline{L'_m}, L_{m+1}, \ldots, L_n \ .$$

3. For every action name $A$, we add the rules

$$\begin{array}{l} A \leftarrow \mathit{not}\ \neg A \ , \\ \neg A \leftarrow \mathit{not}\ A \ , \end{array}$$

and for every fluent name $P$ the rules

$$\begin{array}{l} P \leftarrow \mathit{not}\ \neg P \ , \\ \neg P \leftarrow \mathit{not}\ P \ , \\ \leftarrow \mathit{not}\ P', \mathit{not}\ \neg P' \ . \end{array}$$

According to the answer set semantics, these rules require including one element of each complementary pair $\{A, \neg A\}$ and $\{P, \neg P\}$, and prohibit the answer sets that include neither $P'$ nor $\neg P'$ for some fluent symbol $P$.

Results relating $\mathcal{C}$ to causal logic [Giunchiglia and Lifschitz, 1998] and causal logic to logic programming [Turner, 1998] imply that the answer sets for the program formed in this way from a set of static and dynamic laws are in a 1–1 correspondence with the edges of the transition diagram corresponding to this set.

As an example, consider the logic program that represents propositions (13) and (14). The rules of this program are

$$\begin{array}{l}
P' \leftarrow Q, A \\
P' \leftarrow \mathit{not}\ \neg P', P \\
\neg P' \leftarrow \mathit{not}\ P', \neg P \\
Q' \leftarrow \mathit{not}\ \neg Q', Q \\
\neg Q' \leftarrow \mathit{not}\ Q', \neg Q \\
A \leftarrow \mathit{not}\ \neg A \\
\neg A \leftarrow \mathit{not}\ A \\
P \leftarrow \mathit{not}\ \neg P \\
\neg P \leftarrow \mathit{not}\ P \\
Q \leftarrow \mathit{not}\ \neg Q \\
\neg Q \leftarrow \mathit{not}\ Q \\
\leftarrow \mathit{not}\ P', \mathit{not}\ \neg P' \\
\leftarrow \mathit{not}\ Q', \mathit{not}\ \neg Q'
\end{array} \tag{16}$$

The answer sets

$$\begin{array}{lrrrrrl}
\{ & \neg P, & \neg Q, & \neg A, & \neg P', & \neg Q' & \} \\
\{ & \neg P, & Q, & \neg A, & \neg P', & Q' & \} \\
\{ & P, & \neg Q, & \neg A, & P', & \neg Q' & \} \\
\{ & P, & Q, & \neg A, & P', & Q' & \} \\
\{ & \neg P, & \neg Q, & A, & \neg P', & \neg Q' & \} \\
\{ & \neg P, & Q, & A, & P', & Q' & \} \\
\{ & P, & \neg Q, & A, & P', & \neg Q' & \} \\
\{ & P, & Q, & A, & P', & Q' & \}
\end{array}$$

for this program correspond to the edges of the diagram shown in Figure 2. For instance, the third line from the end corresponds to the vertical edge of the diagram, which begins at the vertex $\overline{P}Q$, is labeled $A$, and ends at $PQ$.

It is interesting to look at the logic programming counterpart of the dynamic laws that express inertia. Consider, for instance, the translation of the first of propositions (14) into logic programming, which is the second rule of (16). This rule says that $P$ remains true after performing an action if there is no evidence that it becomes false. From the perspective of default logic, this is similar to the “frame default” proposed by Reiter [1980] as a solution to the frame problem.

The two constraints at the end of (16) happen to be unnecessary, in the sense that removing them from the program would not affect its answer sets.

The logic programming representation of the static law from the suitcase example, expressed by the third line of (15), is the pair of rules

$$\begin{array}{l} \mathit{Open} \leftarrow \mathit{Up}(L1), \mathit{Up}(L2)\ , \\ \mathit{Open}' \leftarrow \mathit{not}\ \neg \mathit{Up}'(L1), \mathit{not}\ \neg \mathit{Up}'(L2)\ . \end{array}$$

Both occurrences of $\mathit{not}\ \neg$ in the body of the second rule can be actually dropped without changing the answer sets for the program. But there are cases when the presence of negation as failure in the translation of a static law is essential for the soundness of the translation relative to the semantics of $\mathcal{C}$.

# 9 Planning for Domains Described in $\mathcal{C}$

Now we can go back to the issue raised at the beginning of Section 6: the need to extend the planning method from [Subrahmanian and Zaniolo, 1995] and [Dimopoulos *et al.*, 1997] to action domains described using both dynamic and static causal dependencies.

Recall that the translation defined in the previous section gives a logic program whose answer sets represent the edges of the transition diagram for a given set of $\mathcal{C}$ propositions. An easy modification of this translation allows us to represent paths of a fixed length $n$ in this diagram; the original translation corresponds to $n = 1$. We may be able to solve planning problems for domains described in $\mathcal{C}$ by applying a program like SMODELS to the result of this translation.

As an example, consider the following planning problem that involves Lin's suitcase. Initially, both latches are down and the suitcase is closed. In a goal state that we want to achieve, the latches are down but the suitcase is open. If the concurrent execution of toggle actions is allowed then the problem can be solved in two steps: the actions $\mathit{Toggle}(L1)$ and $\mathit{Toggle}(L2)$ are executed concurrently between the times 0 and 1, and then again between the times 1 and 2.

This problem can be submitted to SMODELS as the following input file:

```
up(L,T1) :- latch(L), next(T,T1), nup(L,T), toggle(L,T).
nup(L,T1) :- latch(L), next(T,T1), up(L,T), toggle(L,T).
open(T) :- up(l1,T), up(l2,T).
up(L,T1) :- latch(L), next(T,T1), up(L,T), not nup(L,T1).
nup(L,T1) :- latch(L), next(T,T1), nup(L,T), not up(L,T1).
open(T1) :- next(T,T1), open(T), not nopen(T1).
nopen(T1) :- next(T,T1), nopen(T), not open(T1).

toggle(L,T) :- latch(L), time(T), not last(T), not ntoggle(L,T).
ntoggle(L,T) :- latch(L), time(T), not last(T), not toggle(L,T).
up(L,0) :- latch(L), not nup(L,0).
nup(L,0) :- latch(L), not up(L,0).
open(0) :- not nopen(0).
nopen(0) :- not open(0).

latch(l1).  latch(l2).
time(0).  time(1).  time(2).
last(2).
next(0,1).  next(1,2).

compute 1 {
             not up(l1,0), not up(l2,0), not open(0),
             not up(l1,2), not up(l2,2), open(2)
           }
```

The first group of rules translates propositions (15) into logic programming. The time variables `T`, `T1` and the predicate `next` are used here in place of the distinction between $P$ and $P'$ in the translation from Section 8. The auxiliary predicates `nup` and `nopen` represent the classical negations of `up` and `open`; the need to use these predicates stems from the fact that the current version of SMODELS does not support classical negation.

The two lines in the braces at the end of the input file describe the initial state and the goal of the planning problem.

In the output of SMODELS we read:

```
Stable Model: next(1,2) next(0,1) time(2) time(1) time(0)
latch(l2) latch(l1) nopen(0) nup(l2,0) toggle(l2,0) toggle(l2,1)
nup(l1,0) toggle(l1,0) toggle(l1,1) up(l2,1) up(l1,1) nup(l2,2)
nup(l1,2) open(1) open(2) last(2)
```

The members of this answer set that contain the action name `toggle` represent a solution.

## 10 Conclusion

This paper is a sketch of several streams of ideas that interact in current research on actions and planning: action languages and translating them into logic programming; representing histories by answer sets; satisfiability planning; efficient implementations of the answer set semantics and their use for generating plans; causal logic and its relation to planning algorithms. In the course of this work, researchers have uncovered intriguing connections between the theoretical analysis of causal reasoning on the one hand, and the design of planners on the other.

In the years to come, we will see new advances both in methods for describing action domains and in implemented systems for computing answer sets. This may lead to the creation of planners which are based on the ideas of logic programming and combine the use of expressive action description languages with efficient computational procedures.

## Acknowledgements

Many ideas of this paper have crystallized in the course of stimulating conversations with Michael Gelfond, Norman McCain and Hudson Turner. A talk on this subject was given at the Kentucky meeting on trends and directions in logic programming in April of 1998. I am grateful to the participants of the meeting for useful remarks and criticisms, and especially to Teodor Przymusinski who has presented a commentary on my talk and organized its discussion. This work was partially supported by National Science Foundation under grant IRI-9306751.

## References


[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Denecker and De Schreye, 1993] Mark Denecker and Danny De Schreye. Representing incomplete knowledge in abductive logic programming. In Dale Miller, editor, *Logic Programming: Proceedings of the 1993 Int'l Symposium*, pages 147–163, 1993.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In *Proc. ECP-97*, 1997.

[Dung, 1993] Phan Minh Dung. Representing actions in logic programming and its applications in database updates. In *Logic Programming: Proceedings of the Tenth Int'l Conf. on Logic Programming*, pages 222–238, 1993.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The kr system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.

[Geffner, 1990] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proc. AAAI-90*, pages 524–530, 1990.

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

[Gelfond and Lifschitz, 1993] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, 1998. To appear.

[Green, 1969] Cordell Green. Application of theorem proving to problem solving. In *Proc. IJCAI*, pages 219–240, 1969.

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

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[Lifschitz, 1997] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[Lin, 1995] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. IJCAI-95*, pages 1985–1991, 1995.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 212–223, 1998.

[McCarthy, 1959] John McCarthy. Programs with common sense. In *Proc. Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy*. Ablex, Norwood, NJ, 1990.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, pages 289–303, 1996.

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

[Simons, 1997] Patrik Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report 47, Helsinki University of Technology, 1997.

[Subrahmanian and Zaniolo, 1995] V.S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proc. ICLP-95*, 1995.

[Turner, 1997] Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[Turner, 1998] Hudson Turner. A logic of universal causation. *Artificial Intelligence*, 1998. To appear.