Answer Set Programming and Plan Generation

Vladimir Lifschitz
University of Texas
Austin, TX 78712, USA
vl@cs.utexas.edu

Abstract

The idea of answer set programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver, such as SMODELS or DLV, to find an answer set for this program. Applications of this method to planning are related to the line of research on the frame problem that started with the invention of formal nonmonotonic reasoning in 1980.

1 Introduction

Kautz and Selman [19] proposed to approach the problem of plan generation by reducing it to the problem of finding a satisfying interpretation for a set of propositional formulas. This method, known as *satisfiability planning*, is used now in several planners.[1] In this paper we discuss a related idea, due to Subrahmanian and Zaniolo [36]: reducing a planning problem to the problem of finding an answer set ("stable model") for a logic program. The advantage of this "answer set programming" approach to planning is that the representation of properties of actions is easier when logic programs are used instead of axiomatizations in classical logic, in view of the nonmonotonic character of negation as failure. Two best known answer set solvers (systems for computing answer sets) available today are SMODELS[2] and DLV[3]. The results of computational experiments that use SMODELS for planning are reported in [4, 30].

[1] See `http://www.research.att.com/~kautz/blackbox/` for the latest system of this kind created by the inventors of satisfiability planning.

[2] `http://www.tcs.hut.fi/Software/smodels/` .

[3] `http://www.dbai.tuwien.ac.at/proj/dlv/` .

In this paper, based on earlier reports [22, 23], applications of answer set programming to planning are discussed from the perspective of the research on the frame problem and nonmonotonic reasoning done in AI since 1980. Specifically, we relate them to the line of work that started with the invention of default logic [33]—the nonmonotonic formalism that turned out to be particularly closely related to logic programming [2, 26, 10]. After the publication of the "Yale Shooting Scenario" [14] it was widely believed that the solution to the frame problem outlined in [33] was inadequate. Several alternatives have been proposed [18, 20, 34, 15, 21, 29, 8]. It turned out, however, that the approach of [33] is completely satisfactory if the rest of the default theory is set up correctly [37]. It is, in fact, very general, as discussed in Section 5.2 below. We will see that descriptions of actions in the style of [33, 37] can be used as a basis for planning using answer set solvers.

In the next section, we review the concept of an answer set as defined in [9, 10, 25] and its relation to default logic. Then we describe some of the computational possibilities of answer set solvers (Section 3) and illustrate the answer set programming method [27, 30] by applying it to a graph-theoretic search problem (Section 4). In Section 5 we turn to the use of answer set solvers for plan generation. Section 6 describes the relation of this work to other research on actions and planning.

2 Answer Sets

2.1 Logic Programs

We begin with a set of propositional symbols, called *atoms*. A *literal* is an expression of the form A or $\neg A$, where A is an atom. (We call the symbol $\neg$ "classical negation," to distinguish it from the symbol *not* used for negation as failure.) A *rule element* is an expression of the form L or *not* L, where L is a literal. A *rule* is an ordered pair

$$Head \leftarrow Body \tag{1}$$

where *Head* and *Body* are finite sets of rule elements. A rule (1) is a *constraint* if $Head = \emptyset$; it is *disjunctive* if the cardinality of *Head* is greater than 1. If

$$Head = \{L_1, \dots, L_k, not\ L_{k+1}, \dots, not\ L_l\}$$

and

$$Body = \{L_{l+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n\}$$

($n \geq m \geq l \geq k \geq 0$) then we write (1) as

$$L_1; \ldots; L_k; not\ L_{k+1}; \ldots; not\ L_l \leftarrow L_{l+1}, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n. \tag{2}$$

We will drop $\leftarrow$ in (2) if the body of the rule is empty ($n = m = l$).

A *program* is a set of rules.

These definitions differ from the traditional description of the syntax of logic programs in several ways. First, our rules are propositional: atoms are not assumed to be formed from predicate symbols, constants and variables. An input file given to an answer set solver does usually contain "schematic rules" with variables, but such a schematic rule is treated as an abbreviation for the set of rules obtained from it by grounding. The result of grounding is a propositional object, just like a set of clauses that would be given as input to a satisfiability solver.

On the other hand, in some ways (2) is more general than rules found in traditional logic programs. Each L_i may contain the classical negation symbol $\neg$; traditional logic programs use only one kind of negation—negation as failure. The head of (2) may contain several rule elements, or it can be empty; traditionally, the head of a rule is a single atom. The negation as failure symbol is allowed to occur in the head of a rule, and not only in the body as in traditional logic programming. We will see later that the additional expressivity given by these syntactic features is indeed useful.

2.2 Definition of an Answer Set

The notion of an answer set is defined first for programs that do not contain negation as failure ($l = k$ and $n = m$ in every rule (2) of the program). Let Π be such a program, and let X be a consistent set of literals. We say that X is *closed* under Π if, for every rule (1) in Π, $Head \cap X \neq \emptyset$ whenever $Body \subseteq X$. We say that X is an *answer set* for Π if X is minimal among the sets closed under Π (relative to set inclusion).

For instance, the program

$$\begin{array}{l} p; q\ , \\ \neg r \leftarrow p \end{array} \tag{3}$$

has two answer sets:

$$\{p, \neg r\},\ \{q\}. \tag{4}$$

If we add the constraint

$$\leftarrow q$$

to (3), we will get a program whose only answer set is the first of sets (4). On the other hand, if we add the rule

$$\neg q$$

to (3), we will get a program whose only answer set is $\{p, \neg q, \neg r\}$.

To extend the definition of an answer set to programs with negation as failure, take an arbitrary program Π, and let X be a consistent set of literals. The *reduct* Π^X of Π relative to X is the set of rules

$$L_1; \ldots; L_k \leftarrow L_{l+1}, \ldots, L_m$$

for all rules (2) in Π such that X contains all the literals $L_{k+1}, \ldots, L_l$ but does not contain any of $L_{m+1}, \ldots, L_n$. Thus Π^X is a program without negation as failure. We say that X is an *answer set* for Π if X is an answer set for Π^X.

Consider, for instance, the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q\ , \\ q \leftarrow \mathit{not}\ r\ , \\ r \leftarrow \mathit{not}\ s\ , \end{array} \tag{5}$$

and let X be $\{p, r\}$. The reduct of (5) relative to this set consists of two rules:

$$\begin{array}{l} p\ , \\ r\ . \end{array}$$

Since X is an answer set for this reduct, it is an answer set for (5). It is easy to check that program (5) has no other answer sets.

This example illustrates the original motivation for the definition of an answer set—providing a declarative semantics for negation as failure as implemented in existing Prolog systems. Given program (5), a Prolog system will respond *yes* to a query if and only if that query is p or r, that is to say, if and only if the query belongs to the answer set for (5). In this sense, the role of answer sets is similar to the role of the concept of completion [3], which provides an alternative explanation for the behavior of Prolog (p and r are entailed by the program's completion).

2.3 Comparison with Default Logic

Let Π be a program such that the head of every rule of Π is a single literal:

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n. \tag{6}$$

We can transform Π into a (propositional) default theory in the sense of [33] by turning each rule (6) into the default

$$\frac{L_1 \wedge \cdots \wedge L_m \ : \ \neg L_{m+1}, \ldots, \neg L_n}{L_0}$$

There is a simple correspondence between the answer sets for Π and the extensions for this default theory DT: if X is an answer set for Π then the deductive closure of X is a consistent extension for DT; conversely, every consistent extension for DT is the deductive closure of an answer set for Π.

For instance, the default theory corresponding to program (5) is

$$\frac{:\ \neg q}{p}, \quad \frac{:\ \neg r}{q}, \quad \frac{:\ \neg s}{r}.$$

The only extension for this default theory is the deductive closure of the program's answer set $\{p, r\}$.

Under this correspondence, a rule without negation as failure is represented by a default without justifications, that is to say, by an inference rule. A fact—a rule with the empty body—corresponds to a default that has neither prerequisites nor justifications, that is, an axiom. The normal default

$$\frac{p\ :\ q}{q} \tag{7}$$

is the counterpart of the rule

$$q \leftarrow p, not\ \neg q. \tag{8}$$

Logic programs as defined above are more general than defaults in that their rules may have several elements in the head, and these elements may include negation as failure. On the other hand, defaults are more general in that they may contain arbitrary propositional formulas, not just literals or conjunctions of literals.

In this connection, it is interesting to note that one of the technical issues related to the "Yale Shooting" controversy is whether the effects of actions should be described by axioms, such as

$$loaded(s) \supset \neg alive(result(shoot(s))), \tag{9}$$

or by inference rules, such as

$$\frac{loaded(s)}{\neg alive(result(shoot(s)))}. \tag{10}$$

According to [37], formulation (10) is a better choice. In the language of logic programs (10) would be written as

$$\neg alive(result(shoot(s))) \leftarrow loaded(s).$$

Formula (9), on the other hand, does not correspond to any rule in the sense of logic programming. Paradoxically, limitations of the language of logic programs play a positive role in this case by eliminating some of the "bad" representational choices that are available when properties of actions are described in default logic.

2.4 Generating and Eliminating Answer Sets

From the perspective of answer set programming, two kinds of rules play a special role: those that generate multiple answer sets and those that can be used to eliminate some of the answer sets of a program.

One way to write a program with many answer sets is to use the disjunctive rules

$$A; \neg A \tag{11}$$

for several atoms A. A program that consists of n rules of this form has 2^n answer sets. For instance, the program

$$\begin{array}{l} p; \neg p \ , \\ q; \neg q \end{array}$$

has 4 answer sets:

$$\{p, q\},\ \{p, \neg q\},\ \{\neg p, q\},\ \{\neg p, \neg q\}.$$

As observed in [5], rule (11) can be equivalently replaced in any program by two nondisjunctive rules

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

In the notation of default logic, these rules can be written as

$$\frac{:\ A}{A},\ \frac{:\ \neg A}{\neg A}.$$

Alternatively, a program with many answer sets can be formed using rules of the form

$$L; not\ L \tag{12}$$

where L is a literal. This rule has two answer sets: $\{L\}$ and $\emptyset$. A program that consists of n rules of form (12) has 2^n answer sets—all subsets of the set of literals occurring in the rules. For instance, the answer sets for the program

$$\begin{array}{l} p; \mathit{not}\ p\ , \\ q; \mathit{not}\ q \end{array} \tag{13}$$

are the 4 subsets of $\{p, q\}$.

The rules that can be used to eliminate "undesirable" answer sets are constraints—rules with the empty head. We saw in Section 2.2 that appending the constraint $\leftarrow q$ to program (3) eliminates one of its two answer sets (4). The effect of adding a constraint to a program is always monotonic: the collection of answer sets of the extended program is a part of the collection of answer sets of the original program.

More precisely, we say that a set X of literals *violates* a constraint

$$\leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n \tag{14}$$

if $L_1, \ldots, L_m \in X$ and $L_{m+1}, \ldots, L_n \notin X$. Let Π' be the program obtained from a program Π by adding constraint (14). Then a set X of literals is an answer set for Π' iff

- X is an answer for Π, and
- X does not violate constraint (14).

For instance, the second of the answer sets (4) for program (3) violates the constraint $\leftarrow q$, and the first doesn't; accordingly, adding this constraint to (3) eliminates the second of the program's answer sets.

To see how rules of both kinds—those that generate answer sets and those that eliminate them—can work together, consider the following translation from propositional theories to logic programs. Let Γ be a set of clauses, and let Π be the program consisting of

- rules (11) for all atoms A occurring in Γ, and
- the constraints $\leftarrow \overline{L_1}, \ldots, \overline{L_n}$ for all clauses $L_1 \vee \cdots \vee L_n$ in Γ.

(By $\overline{L}$ we denote the literal complementary to L.) The answer sets for Π are in a 1–1 correspondence with the truth assignments satisfying Γ, every truth assignment being represented by the set of literals to which it assigns the value *true*.

3 Answer Set Solvers

System DLV computes answer sets for finite programs without negation as failure in the heads of rules ($l = k$ in every rule (2) of the program). For instance, given the input file

```
p ; q.
-r :- p.
```

it will return the answer sets for program (3). Given the input file

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

it will return the answer set for program (5).

System SMODELS requires additionally that its input program contain no disjunctive rules. This limitation is mitigated by two circumstances.

First, the input language of SMODELS allows us to express any "exclusive disjunctive rule," that is, a disjunctive rule

$$L_1; \ldots ; L_n \leftarrow Body$$

accompanied by the constraints

$$\leftarrow L_i, L_j, Body \qquad (1 \le i < j \le n).$$

This combination is represented as

$L_1 \mid \; \cdots \; \mid L_n \;\; :- \; Body.$

Second, SMODELS allows us to represent the important disjunctive combination (12) in the head of a rule by enclosing L in braces:

$\{ \; L \; \}.$

A list of rules of the form

$$L_i; not\ L_i \leftarrow Body \qquad (1 \le i \le n)$$

can be conveniently represented in an SMODELS input file by one line

$\{ \; L_1 \;, \; \ldots, \;\; L_n \; \} \;\; :- \; Body.$

For instance, rules (13) can be written simply as

```
{p,q}.
```

Both DLV and SMODELS allow the user to specify large programs in a compact fashion, using rules with schematic variables and other abbreviations. Both systems employ sophisticated grounding algorithms that work fast and simplify the program in the process of grounding.

```
% GENERATE

j {in(X) : vertex(X)}.

% DEFINE

joined(X,Y) :- edge(X,Y).
joined(X,Y) :- edge(Y,X).

% TEST

:- in(X), in(Y),  X!=Y, not joined(X,Y),
   vertex(X), vertex(Y).

% DISPLAY

hide.
show in(X).
```

Figure 1: Search for a large clique.

4 Answer Set Programming

The idea of answer set programming is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to find a solution.

As an example, we will show how this method can be used to find a large clique, that is, a subset V of the vertices of a given graph such that

- every two vertices in V are joined by an edge, and
- the cardinality of V is not less than a given constant j.

Figure 1 shows an SMODELS input file that can be used to find a large clique or to determine that it does not exist. This file is supposed to be accompanied by a file that describes the graph and specifies the value of j, such as the one shown in Figure 2.

The possible values of the variables `X`, `Y` in Figure 1 are restricted by the "domain predicates" `vertex` and `edge`. In case of the graph described in Figure 2, the predicate `vertex` holds for the numerals `0`, ..., `5`, and the

```
const j=3.

vertex(0..5).

edge(0,1).  edge(1,2).  edge(2,0).  edge(3,4).
edge(4,5).  edge(5,3).  edge(4,0).  edge(2,5).
```

Figure 2: A test for the clique program.

predicate `edge` holds for 8 pairs of vertices $\langle 0,1\rangle,\dots,\langle 2,5\rangle$. Accordingly, the expression

```
{in(X) : vertex(X)}
```

at the beginning of Figure 1 ("the set of atoms `in(X)` for all `X` such that `vertex(X)`") has the same meaning as

```
{in(0), in(1), in(2), in(3), in(4), in(5)} .
```

The last expression can be understood as an abbreviation for a set of rules of form (12), as discussed in Section 3. The answer sets for this set of rules are arbitrary sets formed from these 6 atoms. Symbol `j` at the beginning of the rule restricts the answer sets to those whose cardinality is at least `j`. This is an instance of the "cardinality" construct available in SMODELS [31]. It allows the user to bound, from below and from above, the number of atoms of a certain form that are included in the answer set. (A lower bound is placed to the left of the expression in braces, as in this example; an upper bound would be placed to the right.)

The main parts of the program in Figure 1 are the two labeled `GENERATE` and `TEST`. The former defines a large collection of answer sets—"potential solutions." The latter consists of the constraints that "weed out" the answer sets that do not correspond to solutions. As discussed above, a potential solution is any subset of the vertices whose cardinality is at least j; the constraints eliminate the subsets that are not cliques. This is similar to the use of generating and eliminating rules in Section 2.4.

The part labeled `DEFINE` contains the definition of the auxiliary predicate `joined`. The part labeled `DISPLAY` tells SMODELS which elements of the answer set should be included in the output: it instructs the system to "hide" all literals other than those that encode the clique. In case of the problem shown in Figure 2, the part of the answer set displayed by SMODELS is

```
in(5) in(4) in(3) .
```

The discussion of this example in terms of generating a set of potential solutions and testing its elements illustrates the declarative meaning of the program, but it should not be understood as a description of what is actually happening during the operation of an answer set solver. System SMODELS does not process the program shown above by producing answer sets for the `GENERATE` part and checking whether they satisfy the constraints in the `TEST` part, just as a reasonable satisfiability solver does not search for a model of a given set of clauses by generating all possible truth assignments and checking for each of them whether the clauses are satisfied. The search procedures employed in systems SMODELS and DLV use sophisticated search strategies somewhat similar to those used in efficient satisfiability solvers.

Answer set programming has found applications to several practically important computational problems [30, 35, 16]. One of these problems is planning.

5 Planning

5.1 Example

The code in Figures 3–5 allows us to use SMODELS to solve planning problems in the blocks world. We imagine that blocks are moved by a robot with several grippers, so that a few blocks can be moved simultaneously. However, the robot is unable to move a block onto a block that is being moved at the same time. As usual in blocks world planning, we assume that a block can be moved only if there are no blocks on top of it.

There are three domain predicates in this example: `time`, `block` and `location`; a location is a block or the table. The constant `lasttime` is an upper bound on the lengths of the plans to be considered. (To find the shortest plan, one can use the `minimize` feature of SMODELS which is not discussed in this paper.)

The `GENERATE` section defines a potential solution to be an arbitrary set of `move` actions executed prior to `lasttime` such that, for every `T`, the number of actions executed at time `T` does not exceed the number of grippers.

The rules labeled `DEFINE` describe the sequence of states corresponding to the execution of a given potential plan. Each sequence of states is represented by a complete set of `on` literals. The `DEFINE` rules in Figure 5 specify the positive literals describing the initial positions of all blocks. The first two `DEFINE` rules in Figure 3 specify the positive literals describing the po-

```
time(0..lasttime).

location(B) :- block(B).
location(table).

% GENERATE

{move(B,L,T) : block(B) : location(L)} grippers :-
                                      time(T), T<lasttime.

% DEFINE

% effect of moving a block
on(B,L,T+1) :- move(B,L,T),
               block(B), location(L), time(T), T<lasttime.

% inertia
on(B,L,T+1) :- on(B,L,T), not -on(B,L,T+1),
               location(L), block(B), time(T), T<lasttime.

% uniqueness of location
-on(B,L1,T) :- on(B,L,T), L!=L1,
               block(B), location(L), location(L1), time(T).
```

Figure 3: Planning in the blocks world, Part 1.

```
% TEST

% two blocks cannot be on top of the same block
:- 2 {on(B1,B,T) : block(B1)},
   block(B), time(T).

% a block can't be moved unless it is clear
:- move(B,L,T), on(B1,B,T),
   block(B), block(B1), location(L), time(T), T<lasttime.

% a block can't be moved onto a block that is being moved also
:- move(B,B1,T), move(B1,L,T),
   block(B), block(B1), location(L), time(T), T<lasttime.

% DISPLAY

hide.
show move(B,L,T).
```

Figure 4: Planning in the blocks world, Part 2

```
const grippers=2.
const lasttime=3.

block(1..6).

%        Initial state:             Goal:
%
%                                   3   6
%          1   3   5                2   5
%          2   4   6                1   4
%        -------------            ---------

% DEFINE

on(1,2,0).
on(2,table,0).
on(3,4,0).
on(4,table,0).
on(5,6,0).
on(6,table,0).

% TEST

:- not on(3,2,lasttime).
:- not on(2,1,lasttime).
:- not on(1,table,lasttime).
:- not on(6,5,lasttime).
:- not on(5,4,lasttime).
:- not on(4,table,lasttime).
```

Figure 5: A test for the planning program.

sitions of all blocks at time `T+1` in terms of their positions at time `T`. The uniqueness of location rule specifies the negative `on` literals to be included in an answer set in terms of the positive `on` literals in this answer set.

Note that the second `DEFINE` rule in Figure 3 is the SMODELS representation of the normal default

$$\frac{on(b,l,t)\ :\ on(b,l,t+1)}{on(b,l,t+1)} \tag{15}$$

—if it is consistent to assume that at time $t+1$ block b is at the same location where it was at time t then it is indeed at that location (see Section 2.3). This default is interesting to compare with the solution to the frame problem proposed by Reiter in 1980:

$$\frac{R(\mathbf{x},s)\ :\ R(\mathbf{x},f(\mathbf{x},s))}{R(\mathbf{x},f(\mathbf{x},s))}$$

[33, Section 1.1.4]. If we take relation R to be on, and the tuple of arguments $\mathbf{x}$ to be b,l , this expression will turn into

$$\frac{on(b,l,s)\ :\ on(b,l,f(b,l,s))}{on(b,l,f(b,l,s))}. \tag{16}$$

The only difference between defaults (15) and (16) is that the first describes change in terms of the passage of time (t becomes $t+1$), and the latter in terms of state transitions (s becomes $f(b,l,s)$).

Consider now the three constraints labeled `TEST` in Figure 4. The role of the first constraint is to prohibit, indirectly, the actions that would create physically impossible configurations of blocks, such as moving two blocks b_1, b_2 onto the same block b. The other two constraints express the robot's limitations mentioned at the beginning of this section.

Adding these constraints to the program eliminates the answer sets corresponding to the sequences of actions that are not executable in the given initial state. When we further extend the program by adding the `TEST` section of Figure 5, we eliminate, in addition, the sequences of actions that do not lead to the goal state. The answer sets for the program are now in a 1–1 correspondence with solutions of the given planning problem.

The `DISPLAY` section instructs SMODELS to "hide" all literals except for those that begin with `move`. The part of the answer set displayed by SMODELS is the list of actions included in the plan:

```
Stable Model: move(3,table,0) move(1,table,0) move(5,4,1)
move(2,1,1) move(6,5,2) move(3,2,2)
True
Duration: 0.340
```

5.2 Discussion

The description of the blocks world domain in Figures 3 and 4 is more sophisticated, in several ways, than the shooting example [14] that seemed so difficult to formalize in 1987. First, this version of the blocks world includes the concurrent execution of actions.

Second, some effects of moving a block are described here indirectly. In the shooting domain, the effects of all actions are specified explicitly: we are told how the action *load* affects the fluent *loaded*, and how the action *shoot* affects the fluent *alive*. The description of the blocks world given above is different. When block 1, located on top of block 2, is moved onto the table, this action affects two fluents: $on(1, table)$ becomes true, and $on(1, 2)$ becomes false. The first of these two effects is described explicitly by the first `DEFINE` rule in Figure 3, but the description of the second effect is indirect: the uniqueness of location rule allows us to conclude that block 2 is not on top of block 1 anymore from the fact that block 2 is now on the table. The ramification problem—the problem of describing indirect effects of actions—is not addressed in the classical action representation formalisms STRIPS [7] and ADL [32].

Finally, the executability of actions is described in this example indirectly as well. As discussed above, the impossibility of moving two blocks b_1, b_2 onto the same block b is implicit in our description of the blocks world: executing that action would have created a configuration of blocks that is prohibited by one of the constraints in Figure 4. In STRIPS and ADL, the executability of an action has to be described explicitly, by listing the action's preconditions. The usual description of the blocks world asserts, for instance, that moving one block on top of another is not executable if the target location is not clear. This description is not applicable, however, when several blocks can be moved simultaneously: in the initial state shown in Figure 5, block 1 can be moved onto block 4 if block 3 is moved at the same time. Fortunately, when the answer set approach to describing actions is adopted, specifying action preconditions explicitly is unnecessary.

The usefulness of indirect descriptions of action domains for applications of AI was demonstrated in a recent report [38] on modelling the Reaction Control System (RCS) of the Space Shuttle. The system consists of several fuel tanks, oxidizer tanks, helium tanks, maneuvering jets, pipes, valves, and other components. How is the behavior of the RCS affected by flipping one of its switches? According to [38], this action has only one direct effect, which is trivial: changing the position of a switch causes the switch to be in the new position. But there is also a postulate asserting that, if a valve is functional, it is not stuck closed, and the switch controlling it is in the open (or closed) position then the valve is open (or closed). These two facts together tell us that, under certain conditions, flipping a switch indirectly affects the corresponding valve. Furthermore, if a helium talk has correct pressure, there is an open path to a propulsion tank, and there are no paths to a leak, then the propulsion tank has correct pressure also. Using this postulate we can conclude that, under certain conditions, flipping a switch affects pressure in a propulsion tank, and so on. This multi-level approach to describing the effects of actions leads to a well-structured and easy to understand formal description of the operation of the RCS. The answer set programming approach handles such multi-leveled descriptions quite easily.

6 Relation to Action Languages and Satisfiability Planning

Some of the recent work on representing properties of actions is formulated in terms of "high-level" action languages [12], such as $\mathcal{A}$ [11] and $\mathcal{C}$ [13]. Descriptions of actions in these languages are more concise than logic programming representations. For example, the counterparts of the first two `DEFINE` rules from Figure 3 in language $\mathcal{C}$ are

$$move(b,l) \textbf{ causes } on(b,l)$$

and

$$\textbf{inertial } on(b,l).$$

The design of language $\mathcal{C}$ is based on the system of causal logic proposed in [28].

For a large class of action descriptions in $\mathcal{C}$, an equivalent translation into logic programming notation is defined in [24]. The possibility of such a translation further illustrates the expressive power of the action representation method used in this paper.

As noted in the introduction, the answer set programming approach to planning is related to satisfiability planning. There is, in fact, a formal connection between the two methods. If a program without classical negation is "positive-order-consistent," or "tight," then its answer sets can be characterized by a collection of propositional formulas [6]—the formulas obtained by applying the completion process [3] to the program. The translations from language $\mathcal{C}$ described in [24] happen to produce tight programs. Describing a planning problem by a program like this, then translating the program into propositional logic, and, finally, invoking a satisfiability solver to find a plan is a form of satisfiability planning that can be viewed also as "answer set programming without answer set solvers" [1]. This is essentially how planning is performed by the Causal Calculator.[4]

7 Conclusion

In answer set programming, solutions to a combinatorial search problem are represented by answer sets. Plan generation in the domains that involve actions with indirect effects are a promising application area for this programming method.

Systems SMODELS and DLV allow us to solve some nontrivial planning problems even in the absence of domain-specific control information. For larger problems, however, such information becomes a necessity. The possibility of encoding domain-specific control knowledge so that it can be used by an answer set solver is crucial for progress in this area, just as the possibility of using control knowledge by propositional solvers is crucial for further progress in satisfiability planning [17]. This is a topic for future work.

Acknowledgements

Useful comments on preliminary versions of this paper have been provided by Maurice Bruynooghe, Marc Denecker, Esra Erdem, Selim Erdoğan, Paolo Ferraris, Michael Gelfond, Joohyung Lee, Nicola Leone, Victor Marek, Norman McCain, Ilkka Niemelä, Aarati Parmar, Teodor Przymusinski, Mirosław Truszczyński and Hudson Turner. This work was partially supported by the National Science Foundation under grant IIS-9732744.

References

[1] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[5] In *Proc. NMR-2000*, 2000.

[2] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. LICS-87*, pages 89–97, 1987.

[4] `http://www.cs.utexas.edu/users/tag/cc` .

[5] `http://arxiv.org/abs/cs.ai/0003042` .

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

[4] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In Sam Steel and Rachid Alami, editors, *Proc. European Conf. on Planning 1997*, pages 169–181. Springer-Verlag, 1997.

[5] Esra Erdem and Vladimir Lifschitz. Transformations of logic programs related to causality and planning. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 107–116, 1999.

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

[7] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[8] Michael Gelfond. Autoepistemic logic and formalization of commonsense reasoning. In Michael Reinfrank, Johan de Kleer, Matthew Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd Int'l Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 176–186. Springer-Verlag, 1989.

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

[10] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. Seventh Int'l Conf.*, pages 579–597, 1990.

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

[12] Michael Gelfond and Vladimir Lifschitz. Action languages.[6] *Electronic Transactions on AI*, 3:195–210, 1998.

[13] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.

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

[15] Brian Haugh. Simple causal minimizations for temporal persistence and projection. In *Proc. AAAI-87*, pages 218–223, 1987.

[16] Keojo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proc. Fifth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 218–223, 1999.

[17] Yi-Cheng Huang, Bart Selman, and Henry Kautz. Control knowledge in planning: benefits and tradeoffs. In *Proc. AAAI-99*, pages 511–517, 1999.

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

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

[20] Vladimir Lifschitz. Pointwise circumscription: Preliminary report. In *Proc. AAAI-86*, pages 406–410, 1986.

[21] Vladimir Lifschitz. Formal theories of action (preliminary report). In *Proc. IJCAI-87*, pages 966–972, 1987.

[22] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[23] Vladimir Lifschitz. Answer set planning. In *Proc. ICLP-99*, pages 23–37, 1999.

[6] `http://www.ep.liu.se/ea/cis/1998/016/` .

[24] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 92–106, 1999.

[25] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[26] Victor Marek and Mirosław Truszczyński. Stable semantics for logic programs and default theories. In *Proc. North American Conf. on Logic Programming*, pages 243–256, 1989.

[27] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

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

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

[30] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[31] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Extending the stable model semantics. This journal.

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

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

[34] Yoav Shoham. Chronological ignorance: Time, nonmonotonicity, necessity and causal theories. In *Proc. AAAI-86*, pages 389–393, 1986.

[35] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proc. First Int'l Workshop on Practical Aspects of Declarative Languages (Lecture Notes in Computer Science 1551)*, pages 305–319. Springer-Verlag, 1998.

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

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

[38] Richard Watson. An application of action theory to the space shuttle. In Gopal Gupta, editor, *Proc. First Int'l Workshop on Practical Aspects of Declarative Languages (Lecture Notes in Computer Science 1551)*, pages 290–304. Springer-Verlag, 1998.