Applications of Logic Programming to Planning: Computational Experiments

Esra Erdem
Department of Computer Sciences
The University of Texas at Austin, Austin, TX 78712, USA
Email: esra@cs.utexas.edu

March 5, 2000

Abstract

This report summarizes experiments with the systems SMODELS and DLV applied to some planning problems. The planning problems are presented to SMODELS and DLV as logic programs. Plans correspond to answer sets for these programs, in the spirit of answer set programming.

1 Introduction

This report summarizes experiments with the systems SMODELS [Niemelä and Simons, 1996] and DLV [Eiter *et al.*, 1997] applied to some planning problems.

The planning problems are presented to SMODELS and DLV as logic programs. SMODELS and DLV are implementations of answer set semantics. Answer set semantics is introduced by Gelfond and Lifschitz [1990] to provide a semantics for logic programs and is based on the concept of an answer set. For instance, the following logic program Π

$$p \leftarrow not\ q$$
$$q \leftarrow not\ p$$
$$r \leftarrow p$$
$$r \leftarrow q$$

has two answer sets: $\{p, r\}$ and $\{q, r\}$.

The systems we experiment with find some plans that correspond to the answer sets for the given logic program representation of the planning problem in the spirit of [Lifschitz, 1999b]. In this sense, our approach to logic programming is different from the traditional one: instead of asking whether a query is a consequence of a logic program, we ask for the answer sets for the program. This approach is called *answer set programming*, or *stable model programming* [Marek and Truszczyński, 1999].

Our method of encoding planning problems by logic programs is similar to the one used in [Lifschitz, 1999a]. Lifschitz starts with a planning problem described in an expressive language, namely the action language $\mathcal{C}$ from [Giunchiglia and Lifschitz, 1998]. An action language is "a formal model of parts of the natural language that are used for talking about the effects of actions" [Lifschitz, 1999a]. The given planning domain is represented as a transition diagram. The vertices represent the "possible states of the world" and are labeled by functions from the set of "fluents" into the set of "values". The edges represent some triples of the form $\langle s, A, s' \rangle$, where the state s' is a possible "resulting state" of execution of the "action" A in the state s, and are labeled by "actions". A path starting from an initial state and reaching a goal state is a "history of the world" that corresponds to a plan for the given planning problem.

Besides SMODELS and DLV, we experimented with two other systems, CCALC [McCain, 1998] and DERES [Cholewiński *et al.*, 1996b]. CCALC is a system that can reason about actions and find plans using a propositional solver, and DERES is an implementation of default logic.

Our report is organized as follows. In Section 2, we introduce a "suitcase domain" and the first planning problem used in our experiments, and show how to present the problem to DLV and SMODELS. We will present the results of our experiments in Section 3.

Next, we define the Blocks World domain, and several larger planning problems that we experimented with in Section 4. After that, we present the information on computation times in Section 5.

We give a summary of our experimentation with DERES in Section 6, and conclude with a comparison of our approach with related work in Section 7.

2 The Suitcase Problem

The Suitcase domain is introduced in [Lin, 1995]. It is used in [Lifschitz, 1999a] as an example to explain the logic programming approach to planning that we investigate in this report.

2.1 Suitcase Domain

Let's first describe Lin's Suitcase domain: we have a suitcase with two latches $l1$ and $l2$. When these two latches are up then the suitcase automatically opens. There are three propositional fluents: $up(L)$, where L is $l1$ or $l2$, and $open$; $up(L)$ holds iff latch L is up, $open$ holds iff the suitcase is open. There is an action of toggling a latch L denoted by $toggle(L)$. If a latch is down (resp. up) then it becomes up (resp. down) after toggling it.

The transition diagram that corresponds to this domain has 7 possible states:

$$
\begin{array}{ll}
\{up(l1), up(l2), open\} & (S_1) \\
\{up(l1), \neg up(l2), open\} & (S_2) \\
\{up(l1), \neg up(l2), \neg open\} & (S_3) \\
\{\neg up(l1), up(l2), open\} & (S_4) \\
\{\neg up(l1), up(l2), \neg open\} & (S_5) \\
\{\neg up(l1), \neg up(l2), open\} & (S_6) \\
\{\neg up(l1), \neg up(l2), \neg open\} & (S_7)
\end{array}
$$

In every state, an atom is mapped to $false$ if it is preceded by $\neg$, and to $true$ otherwise.

It is important to note that $\{up(l1), up(l2), \neg open\}$ is not a possible state. At each possible state there are 4 applicable actions:

$$
\begin{array}{ll}
\{toggle(l1), toggle(l2)\} & (A_1) \\
\{\neg toggle(l1), \neg toggle(l2)\} & (A_2) \\
\{toggle(l1), \neg toggle(l2)\} & (A_3) \\
\{\neg toggle(l1), toggle(l2)\} & (A_4)
\end{array}
$$

so each state has 4 outgoing edges. For instance, the edges outgoing from S_1 are

$$\langle S_1, A_1, S_6\rangle, \langle S_1, A_2, S_1\rangle, \langle S_1, A_3, S_4\rangle, \langle S_1, A_4, S_2\rangle.$$

The transition diagram that corresponds to the suitcase domain has 28 edges.

The planning problem we experimented with can be described as follows: given

Initial State The latches $l1$ and $l2$ are down, and the suitcase is closed

Goal State The latches $l1$ and $l2$ are down, and the suitcase is open

find a series of actions to reach the goal state from the initial state.

This problem has a solution of length 2 in terms of the transition diagram as follows:

$$\langle S_7, A_1, S_1, A_1, S_6 \rangle. \tag{1}$$

2.2 The Suitcase Problem as a Disjunctive Logic Program

We can describe Lin's Suitcase domain as a disjunctive logic program, i.e., a set of rules of the form[1]

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

Here each L_i is a literal, i.e., an atom possibly preceded by classical negation $\neg$, and $0 \le k \le n \le m$. If $k = 1$ and the body of the rule is empty then the rule is called a *fact*. If $k = 0$ the rule is called a *constraint*.

Our description of Lin's Suitcase domain as a disjunctive logic program is similar to the logic program introduced in [Lifschitz, 1999a]. Programs of this kind can be automatically generated from domain descriptions in $\mathcal{C}$.

If a latch is down (resp. up) then it becomes up (resp. down) after toggling it:

$$\mathit{up}(L, T1) \leftarrow \mathit{latch}(L), \mathit{next}(T, T1), \mathit{toggle}(L, T), \neg \mathit{up}(L, T)$$
$$\neg \mathit{up}(L, T1) \leftarrow \mathit{latch}(L), \mathit{next}(T, T1), \mathit{toggle}(L, T), \mathit{up}(L, T)$$

If both latches are open then the suitcase gets open:

$$\mathit{open}(T) \leftarrow \mathit{up}(l1, T), \mathit{up}(l2, T)$$

The following four rules implement inertia:

$$\mathit{up}(L, T1) \leftarrow \mathit{latch}(L), \mathit{next}(T, T1), \mathit{up}(L, T), \mathit{not}\ \neg \mathit{up}(L, T1)$$
$$\neg \mathit{up}(L, T1) \leftarrow \mathit{latch}(L), \mathit{next}(T, T1), \neg \mathit{up}(L, T), \mathit{not}\ \mathit{up}(L, T1)$$
$$\mathit{open}(T1) \leftarrow \mathit{next}(T, T1), \mathit{open}(T), \mathit{not}\ \neg \mathit{open}(T1)$$
$$\neg \mathit{open}(T1) \leftarrow \mathit{next}(T, T1), \neg \mathit{open}(T), \mathit{not}\ \mathit{open}(T1)$$

Initial values of all fluents are exogenous, that is, any initial value is allowed:

[1] We use ';' (instead of '|' as in [Gelfond and Lifschitz, 1990]) to denote the disjunction in the head of a disjunctive rule.

$$up(L,0); \neg up(L,0) \leftarrow latch(L)$$
$$open(0); \neg open(0) \leftarrow$$

Actions are exogenous:

$$toggle(L,T); \neg toggle(L,T) \leftarrow latch(L), time(T)$$

No actions are executable at the last time:

$$\leftarrow toggle(L,T), lasttime(T)$$

Auxiliary predicates to present the planning problem are defined by the following rules:

$$latch(l1) \leftarrow$$
$$latch(l2) \leftarrow$$
$$time(0) \leftarrow$$
$$time(1) \leftarrow$$
$$time(2) \leftarrow$$
$$next(0,1) \leftarrow$$
$$next(1,2) \leftarrow$$
$$lasttime(2) \leftarrow$$

The answer sets for the above logic program are in a 1-1 correspondence with the paths of length 2 in the transition diagram from Section 2.1. For instance, the answer set that corresponds to path (1) is:

$$\{\neg up(l1,0), \neg up(l2,0), \neg open(0), toggle(l1,0), toggle(l2,0),$$
$$up(l1,1), up(l2,1), open(1), toggle(l1,1), toggle(l2,1),$$
$$\neg up(l1,2), \neg up(l2,2), open(2),$$
$$latch(l1), latch(l2), time(0), time(1), time(2), next(0,1), next(1,2)\}.$$

The problem can be described as finding the answer sets (for the above logic program) that contain

$$\neg up(l1,0), \neg up(l2,0), \neg open(0), \neg up(l1,2), \neg up(l2,2), open(2).$$

2.3 The Suitcase Problem presented to DLV

DLV[2] [Eiter *et al.*, 1997] is a datalog system (i.e., a deductive database system without function symbols) that supports disjunction and classical negation.

A disjunctive logic program presented to DLV is a set of rules of form (2). This program has to satisfy three conditions in order to be presentable to DLV.

The first condition is that the program should not contain any function symbols. This condition guarantees that the Herbrand universe of the program is finite, so that all answer sets for the program are finite too. This is essential for answer set programming systems, which are expected to output all elements of an answer set [Marek and Truszczyński, 1999].

The second condition is that the program should be range-restricted. A logic program Π is *range-restricted* if, for each rule (2) of Π, every variable occurring in the rule also occurs in one of the literals $L_{k+1}, \ldots, L_n$. For instance, the program from Section 2.2 is range-restricted.

The third condition is as follows. An expression of the form P or $\neg P$, where P is a predicate symbol, is *extensional* in a program Π if every rule (2) that contains a literal L_i, $1 \leq i \leq k$, beginning with this expression, is a fact, and *intensional* if no such rule is a fact. For instance, for the program of Section 2.2, $latch$, $time$, and $next$ are extensional expressions, and up, $open$, $toggle$, $\neg up$, $\neg open$, and $\neg toggle$ are intensional expressions. In an input program presented to DLV, every expression of the form P or $\neg P$, where P is a predicate symbol, is required to be extensional or intensional. For instance, the program

$$\begin{array}{l} p(A) \leftarrow \\ p(B) \leftarrow p(C) \end{array}$$

violates this condition. However, the program

$$\begin{array}{l} \neg p(A) \leftarrow \\ p(B) \leftarrow p(C) \end{array}$$

does not violate this condition.

Given a logic program that satisfies the above three conditions, DLV finds the answers sets for the program Π with an algorithm outlined in [Eiter *et al.*, 1998] and explained in [Citrigno *et al.*, 1997] and [Eiter *et al.*, 1997].

Lin's suitcase domain can be presented to DLV as in Figure 1; and the suitcase

[2] There is an online manual for DLV at `http://www.dbai.tuwien.ac.at/proj/dlv/man/`

problem is presented in a separate file as in Figure 2.[3] Then DLV can be invoked by typing

```
dlv suitcase_domain suitcase_problem -n=1 -N=2
```

where `suitcase_domain` is the name of the file presented in Figure 1, and `suitcase_problem` is the name of the file presented in Figure 2. The option `-n=1` specifies that one answer set is to be computed. The option `-N=2` determines the value of `#maxint` (see below) to be 2.

Note that the domain description is similar to the one presented in the previous section. As in Prolog, we use `:-` instead of $\leftarrow$ and include a period at the end of every rule. Classical negation is denoted by '`-`' instead of '$\neg$'.

In the problem description, the initial conditions and the goal are presented as a "query", i.e., an expression of the form

$$b_1, \ldots, b_n, \mathit{not}\ b_{n+1}, \ldots, \mathit{not}\ b_m? \tag{3}$$

where $1 \leq n \leq m$ and each b_i $(1 \leq i \leq m)$ begins with an intensional expression. An answer set *satisfies* query (3) if it contains $b_1, \ldots, b_n$, but not $b_{n+1}, \ldots, b_m$. Given a domain description and a query that describes a problem, DLV finds the answer sets for the domain description that satisfy the query.

A query is a way of representing a set of constraints: including query (3) in the input has the same effect as adding to the program the constraints

$$\begin{array}{l}
\leftarrow \mathit{not}\ b_1 \\
\vdots \\
\leftarrow \mathit{not}\ b_n \\
\leftarrow b_{n+1} \\
\vdots \\
\leftarrow b_m.
\end{array}$$

It is important to note that DLV uses some built-in functions such as `#maxint`, `#int` and `#succ`. Here `#int(T)` expresses that `T` is an integer between 0 and `#maxint`, where the value of `#maxint` is specified in the command line. The atom `#succ(T,T1)` expresses that both `T` and `T1` are between 0 and `#maxint`, and `T1` is `T + 1`. By these built-in functions we can present the applicable time range of the planning problem more elegantly.

[3]Both the domain and the problem for Lin's Suitcase can be introduced in a single file.

```
% effects of toggling
up(L,T1) :- latch(L), next(T,T1), toggle(L,T), -up(L,T).
-up(L,T1) :- latch(L), next(T,T1), toggle(L,T), up(L,T).

% suitcase is open if both of the latches are open
open(T) :- up(l1,T), up(l2,T).

% inertia
up(L,T1) :- latch(L), next(T,T1), up(L,T), not -up(L,T1).
-up(L,T1) :- latch(L), next(T,T1), -up(L,T), not up(L,T1).

open(T1) :- next(T,T1), open(T), not -open(T1).
-open(T1) :- next(T,T1), -open(T), not open(T1).

% initial conditions are exogenous
up(L,0); -up(L,0) :- latch(L).
open(0); -open(0).

% actions are exogenous
toggle(L,T); -toggle(L,T) :- latch(L), time(T).

% no actions are executable at the last time
:- toggle(L,T), lasttime(T).

% auxiliary predicates

latch(l1).
latch(l2).

time(T) :- #int(T).

lasttime(#maxint).

next(X,Y) :- #succ(X,Y).
```

Figure 1: File suitcase_domain for DLV

```
% initial conditions and the goal
-up(l1,0), -up(l2,0), -open(0), open(#maxint), -up(l1,#maxint),
-up(l2,#maxint) ?
```

Figure 2: File `suitcase_problem` for DLV

2.4 The Suitcase Problem presented to SMODELS

SMODELS[4] [Niemelä and Simons, 1996] is an implementation of the answer set ("stable model") semantics for nondisjunctive logic programs.

A logic program that we present to SMODELS should be function-free, as in the case of DLV. Moreover, it is not allowed to contain classical negation $\neg$. Thus in (2) $k \leq 1$ and all literals $L_1, \ldots, L_m$ are atoms.

A logic program presented to SMODELS is also required to be "strongly" range-restricted in the following sense. A predicate symbol P is a *domain predicate* in a program Π, if neither P nor the predicates P depends on use recursion in their definitions. A logic program Π is *strongly range-restricted* if, for each rule (2) of Π, every variable occurring in the rule also occurs in an atom from among $L_{k+1}, \ldots, L_n$ that contains a domain predicate. For instance, the program from Section 2.2 is not strongly range-restricted. Indeed, the predicate *up* is not a domain predicate, because some rules of the program contain *up* both in the head and in the body. Consequently, in the rule

$$open(T) \leftarrow up(l1, T), up(l2, T)$$

the variable T does not occur in an atom containing a domain predicate.

SMODELS takes a function-free strongly range-restricted logic program Π, which contains neither disjunction nor classical negation, and computes the answer sets for the program Π according to the algorithm explained in [Niemelä and Simons, 1996] and [Simons, 1997].

Lin's suitcase domain can be presented to SMODELS as in Figure 3; and the Suitcase problem is presented in a separate file as in Figure 4.[5] Then SMODELS can be invoked by typing

```
lparse -c lasttime=2 suitcase_domain suitcase_problem -c | smodels
```

[4] A manual [Simons, 1996] for SMODELS is available at `http://saturn.hut.fi/pub/smodels/`

[5] Both the domain and the problem for Lin's Suitcase can be introduced in a single file.

where suitcase_domain is the name of the file presented in Figure 3, and suitcase_problem is the name of the file presented in Figure 4. The option -c lasttime=2 specifies that the value of the constant lasttime is 2. By the command lparse the input files are grounded. After that, the answer sets for the ground program are computed by the command smodels.

Let us discuss the differences between Figure 3 and Figure 1. In the input of SMODELS, neither classical negation $\neg$ nor disjunction ; are allowed. Therefore, $\neg up$ and $\neg open$ are replaced in Figures 3 and 4 by new predicates *nup* and *nopen* respectively. Note that there is no predicate that corresponds to $\neg toggle$ in Figures 3 and 4 because the only rules where $\neg toggle$ appears in, which specify that the actions are exogenous, are replaced by

```
{toggle(L,T)} :- latch(L), time(T).
```

By the above rule, any subset Y of the set containing all ground instances of $toggle(L,T)$ is allowed to be contained in an answer set.

The disjunctive rule

$$up(L,0); \neg up(L,0) \leftarrow latch(L)$$

is replaced by

```
up(L,0) | nup(L,0) :- latch(L).
```

which ensures that, for every latch L, either $up(L,0)$ or $nup(L,0)$ is in an answer set. The disjunctive rule, which specifies that $open(0)$ is exogenous, is modified in a similar way.

As the rule

$$open(T) \leftarrow up(l1,T), up(l2,T)$$

is not strongly range-restricted, it is replaced by

$$open(T) \leftarrow time(T), up(l1,T), up(l2,T)$$

In the problem description, the initial conditions and the goal are presented by a compute statement. In a compute statement, we can specify how many answer sets we want SMODELS to find, and which atoms the computed answer sets

```
% effects of toggling
up(L,T1) :- latch(L), next(T,T1), toggle(L,T), nup(L,T).
nup(L,T1) :- latch(L), next(T,T1), toggle(L,T), up(L,T).

% suitcase is open if both of the latches are open
open(T) :- time(T), up(l1,T), up(l2,T).

% inertia
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).

% initial conditions are exogenous
up(L,0) | nup(L,0) :- latch(L).
open(0) | nopen(0).

% actions are exogenous
{toggle(L,T):latch(L)} :- time(T).

% no actions are executed at last time
:- latch(L), toggle(L,lasttime).

% auxiliary predicates
latch(l1).
latch(l2).

time(0..lasttime).

next(T,T+1) :- time(T), lt(T,lasttime).
```

Figure 3: File `suitcase_domain` for SMODELS

```
% initial conditions and the goal
compute 1 {nup(l1,0), nup(l2,0), nopen(0),  open(lasttime),
nup(l1,lasttime), nup(l2,lasttime)}.
```

Figure 4: File suitcase_problem for SMODELS

should or should not contain. For instance, the compute statement in Figure 4 tells SMODELS to find one answer set that contains

$$nup(l1,0), nup(l2,0), nopen(0),$$
$$open(lasttime), nup(l1,lasttime), nup(l2,lasttime).$$

If all were written instead of 1 in the compute statement of Figure 4, all answer sets that contain the above atoms would have been found. Note that in a compute statement the list in braces is a way to represent a set of constraints. In our case including this list has the same effect as adding to the program the constraints

$$\leftarrow not\ nup(l1,0)$$
$$\leftarrow not\ nup(l2,0)$$
$$\leftarrow not\ nopen(0)$$
$$\leftarrow not\ open(lasttime)$$
$$\leftarrow not\ nup(l1,lasttime)$$
$$\leftarrow not\ nup(l2,lasttime).$$

SMODELS does not have built-in functions such as #maxint, #int, and #succ of DLV. However, it has built-in function + and operator lt used in

```
next(T,T+1) :- time(T), lt(T,lasttime).
```

where the atom lt(T,lasttime) expresses that T is less than lasttime. In addition, it has an elegant way of representing a range of integers as in

```
time(0..lasttime).
```

3 Experimental Evaluation for the Suitcase Problem

The experiments are performed on an UltraSPARC[6] with the files presented in Figures 1–4.

In the command line

```
time dlv suitcase_domain suitcase_problem -n=1 -N=2
-pfilter=toggle
```

we use the shell command "time" to get the overall computational time. The option `-pfilter=toggle` instructs DLV to exclude from its output all literals other than the atoms beginning with *toggle*. As a result, the output will show the plan only, instead of the whole answer set.

DLV finds a plan for the Suitcase problem as follows:

```
dlv [build BEN/Nov 24 1999   gcc egcs-2.91.66 19990314
(egcs-1.1.2 release)]

Reading "suitcase_domain"
Reading "suitcase_problem"

{toggle(l1,0), toggle(l2,0), toggle(l1,1), toggle(l2,1)}
0.04u 0.01s 0:00.05 100.0%
```

In the above output, the first two lines give information about the version of DLV being run, and the compiler being used. The next two lines show the names of the input files. After that, the plan found for the planning problem described in the input files is presented.

From the last line of the above output, we can find that the elapsed time is 0.04 + 0.01 seconds where `0.04u` denotes the user time, `0.01s` denotes the system time, and `0:00.05` denotes the overall time in CPU seconds.

If you want to see only the plan found by DLV and the duration, `-silent` may be added as an option in the command line.

We may get the shortest plan for a given planning problem using the script file `smallest-plan` which calls DLV repeatedly incrementing N by 1 at each iteration where the initial value of N can be stated in the command line. The default initial value of N is 0.

[6]The UltraSPARC used for our experiments has 124 MB main memory, runs SunOS 5.5.1, and has a 167 MHz CPU.

We type the following command line to invoke SMODELS:

```
time sm_lparse suitcase_domain suitcase_problem 2.
```

In this command line, "time" is used to compute the duration of overall computation, i.e., grounding and model-finding. The command `sm_lparse` is the name of a script file that contains

```
#!/usr/bin/csh
lparse -d none -c lasttime=$3 $1 $2 | smodels
```

The reason we use the above script is to gather the command lines required to invoke SMODELS into one command line. In this way, the duration of overall computation is found by one command line[7]. In this command line, the input files are "intelligently" grounded by SMODELS' parser `lparse` in the command line `lparse -d none -c lasttime=$3 $1 $2`. All of the domain predicates are dropped from the rules of the ground program by the option `-d none`. After grounding the input files, the answer sets for the ground program are found by the command `smodels`.

In order to exclude from SMODELS's output the atoms other than the ones beginning with *toggle*, we add to the program presented in Figure 4 the following lines:

```
hide open(T).
hide nopen(T).
hide up(L,T).
hide nup(L,T).
```

Then SMODELS finds a plan for the Suitcase problem as follows:

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: toggle(l2,0) toggle(l1,0) toggle(l2,1) toggle(l1,1)
True
Duration: 0.010
Number of choice points: 0
Number of wrong choices: 0
```

[7]It is important to note that the time spent for pipelining is also included in overall computation time.

```
Number of atoms: 31
Number of rules: 46
Number of picked atoms: 2
Number of forced atoms: 1
Number of truth assignments: 46
Size of searchspace (removed): 0 (0)
0.12u 0.05s 0:00.19 89.4%
```

In the above output, the first line gives some information about the version of SMODELS being run. The second line tells that one answer set is found. This answer set is displayed in the next three lines.

In the above output, some information about the existence of other answer sets is also given. For instance, `True` appearing just before the statistics means that there may be another answer set. If we have asked SMODELS to find all of the answer sets for the Suitcase problem, we would get `False` instead of `True`.

In addition, the duration of model-finding, and some other detailed information are shown as well. They are explained in [Simons, 1996], [Simons, 1997] and [Niemelä and Simons, 1996].

4 Blocks World Problems

In the Blocks World domain, we have blocks $b_1, b_2, \ldots, b_n$ on a table. The fluent $on(B, L)$ expresses that block B is at location L, where L can be a block or the table, and the fluent $supported(B)$ expresses that block B is supported by the table. There is an action of moving a block B onto a location L denoted by $move(B, L)$. In this domain, the actions are not allowed to occur concurrently. We can describe the blocks world domain as a disjunctive logic program as follows.

The following rule describes the effect of moving a block:

$$on(B, L, T1) \leftarrow block(B), location(L), move(B, L, T), next(T, T1)$$

A block can be moved only when it's clear:

$$\leftarrow move(B, L, T), on(B1, B, T)$$

Any two blocks cannot be on the same block at the same time:

$$\leftarrow block(B), on(B2, B, T), on(B1, B, T), not\ eq(B1, B2)$$

Wherever a block is, it's not anywhere else:

$$\neg on(B, L1, T) \leftarrow on(B, L, T), location(L1), not\ eq(L, L1)$$

Every block is supported by the table:

$$supported(B, T) \leftarrow on(B, table, T),$$
$$supported(B, T) \leftarrow on(B, B_1, T), supported(B_1, T), not\ eq(B, B1),$$
$$\leftarrow not\ supported(B, T).$$

No actions are executed at last time:

$$\leftarrow move(B, L, T), lasttime(T)$$

No concurrency is allowed:

$$\leftarrow move(B, L, T), move(B1, L1, T), B < B1$$
$$\leftarrow move(B, L, T), move(B1, L1, T), L < L1$$

The following rule implements inertia:

$$on(B, L, T1) \leftarrow on(B, L, T), next(T, T1), not\ \neg on(B, L, T1)$$

The initial values and actions are exogenous:

$$on(B, L, 0); \neg on(B, L, 0) \leftarrow block(B), location(L)$$
$$move(B, L, T); \neg move(B, L, T) \leftarrow block(B), location(L), time(T)$$

Time is defined as follows:

$$time(0) \leftarrow$$
$$time(1) \leftarrow$$
$$time(2) \leftarrow$$
$$time(3) \leftarrow$$
$$next(0, 1) \leftarrow$$
$$next(1, 2) \leftarrow$$

$next(2,3) \leftarrow$

Location is defined as follows:

$location(L) \leftarrow block(L)$
$location(table) \leftarrow$
$eq(L,L) \leftarrow location(L)$

In our experiments, we presented the no-concurrency constraints to SMODELS by:

```
:- 2{move(B,L,T):block(B):location(L)}, time(T).
```

By the above constraint, the answer sets containing any subset, with cardinality greater than or equal to 2, of the set containing all ground instances of $move(B,L,T)$ for time T are eliminated.

First we experimented with the Sussman anomaly [Sussman, 1990]: given

```
Initial State:                 Goal State:

                                  b2
  b2                              b1
  b0     b1                       b0
------------                 -------------
```

find a series of actions to reach the goal state from the initial state.

To describe the Sussman anomaly we define the predicate *block* as follows:

$block(b_0) \leftarrow$
$block(b_1) \leftarrow$
$block(b_2) \leftarrow$

and we specify the lasttime to be 3. The problem can be described as finding an answer set that contains

$$on(b_1, table, 0), on(b_2, b_0, 0), on(b_0, table, 0),$$
$$on(b_0, table, 3), on(b_1, b_0, 3), on(b_2, b_1, 3).$$

Besides the Sussman anomaly, we experimented with the blocks world problems presented in Figure 5.

P1

```
 Initial State:                Goal State:

                                   b3
                                   b2
              b3                   b1
  b0    b1    b2                   b0
-----------------             -------------
```

P2

```
 Initial State:                Goal State:

                                   b4
                                   b3
          b0                       b2
    b1    b3                       b1
    b4    b2                       b0
  -------------               -------------
```

P3

```
 Initial State:                Goal State:

  b2          b7                   b7          b5
  b3    b4    b6                   b3    b2    b0
  b0    b1    b5                   b4    b6    b1
------------------             -------------------
```

P4

```
 Initial State:                Goal State:

                                               b1
        b10   b8                               b2
  b2    b9    b7                   b0    b7    b10
  b1    b4    b6                   b4    b8    b6
  b0    b3    b5                   b9    b3    b5
------------------             --------------------
```

Figure 5: The blocks world problems we experimented with

5 Experimental Evaluation for Blocks World Problems

Both systems SMODELS and DLV find the following plan for the Sussman anomaly with 3 moves:

$$move(b_2, table, 0),\ move(b_1, b_0, 1),\ move(b_2, b_1, 2).$$

For P1, SMODELS finds the following plan with 4 moves:

$$move(b_1, b_0, 0),\ move(b_3, table, 1),\ move(b_2, b_1, 2),\ move(b_3, b_2, 3)$$

and DLV finds the following plan with 4 moves:

$$move(b_3, table, 0),\ move(b_1, b_0, 1),\ move(b_2, b_1, 2),\ move(b_3, b_2, 3).$$

Both systems find a plan with 6 moves for P2:

$$move(b_0, table, 0),\ move(b_3, table, 1),\ move(b_1, b_0, 2),\ move(b_2, b_1, 3),\ move(b_3, b_2, 4),\ move(b_4, b_3, 5).$$

For P3, SMODELS finds the following plan with 8 moves:

$$move(b_7, table, 0),\ move(b_6, table, 1),\ move(b_2, b_6, 2),\ move(b_4, table, 3),\ move(b_3, b_4, 4),\ move(b_7, b_3, 5),\ move(b_0, b_1, 6),\ move(b_5, b_0, 7)$$

and DLV finds the following plan with 8 moves:

$$move(b_4, table, 0),\ move(b_7, b_1, 1),\ move(b_6, table, 2),\ move(b_2, b_6, 3),\ move(b_3, b_4, 4),\ move(b_7, b_3, 5),\ move(b_0, b_1, 6),\ move(b_5, b_7, 7).$$

SMODELS finds the following plan for P4 with 9 moves:

$$move(b_{10}, b_2, 0),\ move(b_9, table, 1),\ move(b_4, b_9, 2),\ move(b_8, b_3, 3),\ move(b_7, b_8, 4),\ move(b_{10}, b_6, 5),\ move(b_2, b_{10}, 6),\ move(b_1, b_2, 7),\ move(b_0, b_4, 8).$$

and DLV finds the following plan for P4, with 9 moves:

$move(b_{10}, table, 0)$, $move(b_9, table, 1)$, $move(b_4, b_9, 2)$,
$move(b_8, b_3, 3)$, $move(b_7, b_8, 4)$, $move(b_{10}, b_6, 5)$,
$move(b_2, b_{10}, 6)$, $move(b_1, b_2, 7)$, $move(b_0, b_4, 8)$.

We experimented with DLV on two different logic program representations of Blocks World problems:

(a) with rules

$$on(B, L, 0); \neg on(B, L, 0) \leftarrow block(B), location(L) \tag{4}$$

$$move(B, L, T); \neg move(B, L, T) \leftarrow block(B), location(L), time(T) \tag{5}$$

(b) with both (4) and (5) replaced by nondisjunctive rules

$$\begin{array}{l} on(B, L, 0) \leftarrow block(B), location(L), not\ \neg on(B, L, 0) \\ \neg on(B, L, 0) \leftarrow block(B), location(L), not\ on(B, L, 0) \\ move(B, L, T) \leftarrow block(B), location(L), time(T), not\ \neg move(B, L, T) \\ \neg move(B, L, T) \leftarrow block(B), location(L), time(T), not\ move(B, L, T) \end{array}$$

The experiments are performed on the same UltraSparc as with the Suitcase problem.

For comparison, we also experimented with CCALC, and we want to present the results of these experiments as well. CCALC is a system that can reason about actions and find plans using a propositional solver such as SATO [Zhang, 1997]. Its input can be represented in the action language $\mathcal{C}$ [Giunchiglia and Lifschitz, 1998], which is what we did in our experiments. CCALC can transform its input into a propositional theory in clausal form in three ways: using the *basic mode*, using the *history mode* or using the *transition mode*. The transition mode differs in that CCALC first constructs a propositional theory describing the effects of actions at time 0. It converts this classical propositional theory into clausal form and then "shifts" the clauses to generate similar descriptions for other time instants.[8]

After transforming the causal theory to a propositional theory in clausal form, CCALC can sort the clauses in this theory. In our experiments, clauses are not sorted. Then CCALC uses a propositional solver to find a model of this classical

[8]The computation times in the table below correspond to the transition mode.

propositional theory. This process can be used, in particular, for finding plans in the spirit of satisfiability planning [Kautz and Selman, 1992]. These models correspond to the answer sets for the logic programs from which the classical propositional theories produced by CCALC can be obtained by essentially applying completion [Clark, 1978]. This is discussed in more detail in Section 7.2.

The completion of the blocks world description presented to SMODELS has models that do not correspond to valid configurations of blocks such as two blocks being on top of each other. It is explained in [Lifschitz, 1999b] that these models can be eliminated by replacing the rules

$$supported(B,T) \leftarrow on(B,table,T),$$

$$supported(B,T) \leftarrow on(B,B_1,T), supported(B_1,T), not\ eq(B,B1),$$

$$\leftarrow not\ supported(B,T).$$

by the rules

$$above(B,L,T) \leftarrow on(B,L,T),$$

$$above(B,L,T) \leftarrow on(B,B_1,T), above(B_1,L,T),$$

$$\leftarrow above(B,B,T),$$

$$\leftarrow not\ above(B,table,T).$$

If we modify the program introduced in Section 4 in the same way then, in our experiments, the models of the classical propositional theory produced by CCALC correspond to the answer set sets for that program.

We used CCALC 1.23 with SATO 3.2, SMODELS 2.5 with LPARSE 0.99.49, and DLV November 24, 1999 Version in our experiments. The following table summarizes the duration of the computations (in CPU seconds) for each problem applied to these three systems[9]:

[9] It is important to keep in mind that grounding and conversion to clauses in CCALC is implemented in Prolog, and SMODELS and DLV are implemented in C++.

Problem	number of blocks	length of plan	CCALC (transformation + SATO)	SMODELS (LPARSE + SMODELS)	DLV	
					(a)	(b)
P0	3	3	0.36 + 0.01	0.1 + 0.12	0.19	0.19
P1	4	4	0.83 + 0.05	0.2 + 0.2	0.49	0.49
P2	5	6	1.65 + 0.15	0.4 + 0.49	1.52	1.52
P3	8	8	7.03 + 1.96	1.39 + 10	88.56	92.88
P4	11	9	20.96 + 12.17	3.52 + 19.63	299.03	301.79

The transformation phase of CCALC takes more time in the history mode. For instance, the transformation phase takes 31.38 seconds for P3 and 105.41 seconds for P4 in the history mode.

If the clauses are sorted then the time spent in model-finding may decrease. However, sorting the clauses may take more time than what is gained in model-finding. For instance, for P4, SATO spends 8.12 seconds when the clauses are sorted whereas sorting the clauses takes 12.8 seconds.

If we look at the time spent at each phase of the computation, we can see that CCALC spends most of the elapsed time for transforming its input into an equivalent classical propositional theory while SMODELS spends most of the elapsed time for finding an answer set. For instance, for P4, CCALC spends 20.96 seconds for grounding, transforming the causal theory into propositional theory, conversion to the clausal form, and clause-shifting, and 12.17 seconds in model-finding. If we measure the time spent in grounding and model-finding separately for SMODELS, we find out that SMODELS spends 3.52 seconds in grounding, and 19.63 seconds in model-finding for P4. SMODELS spends less time in grounding as it makes use of the given query to produce a sufficient subset of the ground instances of the given program. SMODELS can eliminate the domain predicates (depending on the command line option `-d`) appearing in the ground program so that a smaller ground program is used to find the answer sets. This makes SMODELS more efficient in computing the answer sets after grounding.

We don't have enough information about the time spent during each phase of computation in DLV.

DLV, unlike CCALC and SMODELS, uses a general optimization technique [Faber *et al.*, 1999] to produce a smaller ground program. For instance, with this technique, the no-concurrency constraints

$$\leftarrow move(B,L,T), move(B1,L1,T), B < B1,$$
$$\leftarrow move(B,L,T), move(B1,L1,T), L < L1$$

are replaced by

$$
\begin{array}{l}
move1(B,T) \leftarrow move(B,L,T), \\
\leftarrow move1(B,T), move1(B1,T), B < B1, \\
move2(L,T) \leftarrow move(B,L,T), \\
\leftarrow move2(L,T), move2(L1,T), L < L1.
\end{array}
$$

Note that the number of ground instances of the no-concurrency rules is reduced.

6 Experimentation with DERES

We also experimented with the system DERES [Cholewiński *et al.*, 1996a], an implementation of default logic, on the Suitcase problem.

Default logic is a knowledge representation formalism invented by Reiter [1980]. It is a nonmonotonic extension of first-order logic. A default theory is a pair (D, W), where D is a set of "defaults" and W is a set of formulas–axioms. Defaults are expressions of the form

$$\frac{A : B_1, \ldots, B_n}{C}$$

where A, B_i, and C are formulas of first-order logic. Historically, Reiter's semantics of defaults was an inspiration for the concept of an answer set. Indeed, a nondisjunctive logic program rule

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

can be identified with the default

$$\frac{L_1 \wedge \ldots \wedge L_m : \neg L_{m+1}, \ldots, \neg L_n}{L_0}. \tag{6}$$

Notice that $not\ L$ in a logic program corresponds to $\neg L$ after the colon in a default. There is a 1-1 correspondence between answer sets for a logic program and extensions for the corresponding default theory. The answer sets for a logic program can be obtained from the extensions for the corresponding default theory by intersecting them with the set of literals.

In our experiments, we used the Suitcase problem described as a nondisjunctive normal logic program (as shown in Figures 3 and 4). This logic program is transformed to a set of defaults is by the program SM2DT. This transformation is similar to the one described above.

The program SM2DT converts a nondisjunctive normal logic program Π to a default theory (D, W) as follows. First, it calls LPARSE, with Π as the input, to produce the ground version of Π in SMODELS 1 format. Here LPARSE changes the order of the rules in Π. Then it calls the program SIMPLIFY, with the output of LPARSE, to replace the "compute" statement and the constraints by a set of rules, and do some "simplifications". For instance, if the compute statement is

```
compute 1 {p, not q}.
```

SIMPLIFY removes every rule that contains `not p` or `q` in its body, removes `not q` from the bodies of the remaining rules, adds the rule

```
Gpr00000 :- not p, not Gpr00000.
```

where `Gpr00000` is invented by SIMPLIFY, and appends `not q` to the body of every rule with the head `q`. SIMPLIFY also replaces the constraint

```
:- not q.
```

by the rule

```
_false :- not q, not _false.
```

where `_false` denotes $\perp$.

Next, SM2DT calls the program STRATIFY, with the output of SIMPLIFY. STRATIFY reorders the rules in this program and finds a "stratification" (or a "splitting") for that program. The stratification information significantly improves the computational efficiency of DERES. For instance, for stratify reorders the rules of the program

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

as follows:

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

and finds the following stratification

```
2
1
```

describing that the output program of STRATIFY can be splitted by a set of literals such that the first two rules form the "bottom", and the last one forms the "top" of the program.

Finally, it calls the program SMDTCONV with the outputs of STRATIFY to produce a default theory that can be presented to DERES. SMDTCONV transforms the rules of the logic program output of STRATIFY to a set of defaults D, as described above. For instance, the rule

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

is transformed to the default

```
 : !q
      -> p;
```

in DERES format. Here, `!` denotes the classical negation $\neg$. SMDTCONV defines W to be the empty set.

After SM2DT converts a nondisjunctive normal logic program to a default theory, we use SDERES, a version of DERES, with this default theory to find its extensions. These extensions are in 1-1 correspondence with the answer sets for the nondisjunctive normal logic program given as input to SM2DT.

All of the programs mentioned above are contained in a package named LPSMS. The version of LPSMS we used is 0.5.1. It contains LPARSE 0.99.49 as well. By the following directive:

```
scripts/sm2dt -d conv/ -p  'lparse -1 -d none -c lasttime=2'
-o suitcase suitcase_domain suitcase_problem
```

SM2DT displays the following:

```
Parsing (using lparse -1 -d none -c lasttime=2)...
Simplifying...
simplification: 0.000000 seconds
Stratifying...
stratification: 0.010000 seconds
Translating...
conversion: 0.000000 seconds
Cleaning up...
```

and produces the files `suitcase.dt` (containing the names of the files that contain the set of defaults and the set of axioms), `suitcase.dc` (containing the set of defaults), `suitcase.thc` (containing the set of axioms) and `suitcase.str` (containing the stratification). By the following directive:

```
sderes suitcase
```

SDERES finds the folowing extension:

```
Extension 1: Cn(W + {nup(l1,0), nup(l2,0), nopen(0), ntoggle(l1,2),
toggle(l1,1), toggle(l1,0), up(l1,1), nup(l1,2), ntoggle(l2,2),
toggle(l2,1), toggle(l2,0), up(l2,1), nup(l2,2), open(1), open(2) })

1 extension
0.010000 seconds
```

7 Comparisons with Related Work

In our experiments, we specified the planning problems with logic programs. As we emphasized in Section 1, our approach to logic programming is answer set programming. In the next subsection, we will compare our experiments with other experiments that apply answer set programming to planning. In the last subsection, we will compare two approaches, answer set programming and satisfiability checking, used to solve the planning problems.

7.1 Encodings of the Planning Problems

The earliest use of answer set programming for planning is described by Dimopoulos, Nebel, and Koehler [1997]. The encodings of planning problems used in [Lifschitz, 1999a] and in this report are different from theirs in three ways.

First, in [Dimopoulos *et al.*, 1997], the authors start with planning problems described in STRIPS representation [Fikes and Nilsson, 1971]; we are interested in planning problems described in a more expressive language, namely the action language $\mathcal{C}$, as in [Lifschitz, 1999a].

Second, to encode the planning problems in a more compact and efficient way, they use linear encodings (introduced by Kautz and Selman [1992]), i.e., splitting the action predicates into a number of binary predicates. Our encodings are more natural and straightforward but less efficient.

The third difference is as follows. In some of the problems, the actions are allowed to be executed concurrently, as in the Suitcase problem. In some other problems, such as the Blocks World problems above, the parallel execution of actions is not allowed. In such problems, it may be easier to find a plan with concurrently executed actions and then "serialize" it. This is what Dimopoulos, Nebel, and Koehler do for planning in the Blocks World. We do not use any post-processing methods to make planning more efficient. We try to present the planning problems in a form that can be automatically generated from the problem description in the action language $\mathcal{C}$ as in [Lifschitz, 1999a].

The paper [Niemelä, 1998] describes further experiments similar to those done by Dimopoulos *et al.*.

7.2 Answer Set Planning vs. Satisfiability Planning

In our approach to planning, a plan corresponds to an answer set for the logic program representation of the planning problem in the spirit of [Lifschitz, 1999b]. In satisfiability planning [Kautz and Selman, 1992], a plan corresponds to a model satisfying a set of propositional formulas describing the planning problem.

Every answer set for a logic program Π is a model of the completion of Π, but not the other way around. "Translating" from the language of propositional logic programs into the language of propositional formulas can be accomplished by the process of completion [Clark, 1978]. However, there is a special class of programs for which they do [Fages, 1994], [Lifschitz, 1996]: finite "tight" logic programs that do not contain classical negation. With the generalized version of Fages' theorem [Babovich *et al.*, 2000], this is applicable to the Blocks World problems we experimented with.

The system CCALC [McCain, 1998], [McCain and Turner, 1998] reasons about actions and finds plans using a satisfiability checker. Currently, it can use two checkers: RELSAT [Bayardo and Schrag, 1997] or SATO [Zhang, 1997]. The computational procedure used by CCALC, called "literal completion", is a modification of Clark's completion [Clark, 1978] applicable to programs with classical negation which is due to McCain and Turner [1997].

We experimented with CCALC (using SATO) in order to have an idea of the time performance of the systems using answer set programming compared to the system using satisfiability checking, and presented the computation times in Section 5.

Acknowledgments

Thanks to Vladimir Lifschitz for his advice and support. Thanks to Norman McCain for his help in CCALC, Gerald Pfeifer, Nicola Leone and Wolfgang Faber for their help in DLV, Pawel Cholewinski, Artur Mikitiuk, Neil Moore and Miroslaw Truszczyński for their help in DERES, and Ilkka Niemelä and Patrik Simons for their help in SMODELS. Thanks to Norman McCain and Hudson Turner for useful discussions related to representing blocks world domain as a logic program and as a causal theory. Thanks to Enrico Giunchiglia for useful discussions about the satisfiability solvers. Thanks to Hantao Zhang for his help in SATO. Thanks also to Alessandro Provetti for his comments on a draft of this report. This work was partially supported by National Science Foundation under grant IIS-9732744.

References

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming, 2000. Unpublished Draft.

[Bayardo and Schrag, 1997] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97*, pages 203–208, 1997.

[Cholewiński *et al.*, 1996a] Pawel Cholewiński, Victor W. Marek, Artur Mikitiuk, and Miroslaw Truszczyński. Computing with default logic, 1996.

[Cholewiński *et al.*, 1996b] Pawel Cholewiński, Victor W. Marek, and Miroslaw Truszczyński. Default reasoning system DERES. In J. Doyle L. Carlucci Aiello and S. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 518 – 528, 1996.

[Citrigno *et al.*, 1997] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The DLV system: Model generator and application frontends. In *Proceedings of Workshop on Logic Programming (WLP97)*, 1997.

[Clark, 1978] Keith Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer-Verlag, 1997.

[Eiter *et al.*, 1997] Thomas Eiter, Nicola Leone, Christinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for non-monotonic reasoning. In *Proceedings of the 4th International Conference on Logic Programmimg and Nonmonotonic Reasoning*, pages 363–374. Springer-Verlag, 1997.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Christinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: progress report, comparisons and benchmarks. In *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[Faber *et al.*, 1999] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming*, 1999.

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

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: a new approach to the application of the theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gelfond and Lifschitz, 1990] 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.

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

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

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pages 69–127, 1996.

[Lifschitz, 1999a] Vladimir Lifschitz. Action languages, answer sets and planning. *25 Years of Logic Programming: Past and Future*, 1999.

[Lifschitz, 1999b] Vladimir Lifschitz. Answer set planning. In *Proc. ICLP-99*, 1999. To appear.

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

[Marek and Truszczyński, 1999] Victor Marek and Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. *25 Years of Logic Programming: Past and Future*, 1999.

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

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In *Proceedings of Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 212–223, 1998.

[McCain, 1998] Norman McCain. A manual for the causal calculator, 1998.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[Niemelä, 1998] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79. Helsinki University of Technology, Digital Systems Laboratory, Research Report A52, 1998.

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

[Simons, 1996] Patrik Simons. Computing stable models, 1996.

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

[Sussman, 1990] J. Gerald Sussman. The virtuous nature of bugs. *Readings in Planning*, pages 11–117, 1990.

[Zhang, 1997] H. Zhang. SATO: An efficient propositional prover. In *International Conference on Automated Deduction (CADE'97)*, 1997.