Using Stable Model Semantics (SMODELS) in the Causal Calculator (CCALC)

Semra Doğandağ*, F. Nur Alpaslan*, Varol Akman**

* Department of Computer Engineering
Middle East Technical University
Ankara, TR-06531 Turkey
{semra, alpaslan}@ceng.metu.edu.tr
** Department of Computer Engineering
Bilkent University
Ankara, TR-06533 Turkey
akman@cs.bilkent.edu.tr

ABSTRACT

Action Languages are formal methods of talking about actions and their effects on fluents. One recent approach in planning is to define the domains of the planning problems using action languages. The aim of this research is to find a plan for a system defined in the action language *C* by translating it into a causal theory and then finding an equivalent logic program. The planning problem will then be reduced to finding the answer set (stable model) of this logic program. This planner will be added as an extension to the Causal Calculator (CCALC) which is a model checker for the language of the causal theories.

I INTRODUCTION

Plan generation plays an important role in AI research. A lot of work has been done on representing actions and generating plans. While reasoning about action, some problems (such as the frame, ramification, and qualification problems) have to be taken into account. It has been shown [3] that these problems can be overcome to some extent by using causal knowledge. CCALC, which is a system written at the University of Texas, Austin, finds plans and reasons about the actions defined in causal theories. Programs written in the action description language *C* can be given as input to the CCALC system. CCALC translates them to the corresponding causal theory. The *C* language can express indirect effects, implicit preconditions, action preconditions, nondeterminism, concurrent actions, and noninertial fluents. CCALC gets the equivalent propositional representation of the problem by literal completion and finds the models of the system by using the available satisfiability solvers. (A user of CCALC currently has the choice of using either "relsat" [13] or "sato" [4] as the satisfiability solver).

Another perspective to planning is to use logic programming. The language of logic programming offers a reasonably expressive way, enabling us to describe the effects of actions. Also there is a relation between logic programming and causal theories in that both are *noncontrapositive*[1]. As is mentioned in [17] the difference between

$$p \leftarrow not\ q \tag{1}$$

and

$$q \leftarrow not\ p \tag{2}$$

is apparent both in logic programming and in causal theories but is lost if propositional logic is used. The first sentence means that the absence of *q* is a cause for *p*, but this doesn't mean that the absence of *p* is a cause for *q*, which is what the second sentence says. To combine the ideas of logic programming with the use of expressive action description languages, the domain described in the action language can be translated into an equivalent logic program. Our research will realize this by first translating a program in *C* language to a causal theory and then the causal theory to the corresponding logic program. The answer set of the program will be used for planning or querying the plans found.

Causality is an important concept for nonmonotonic reasoning while working on action representations and their effects on the fluents. Situation calculus which specifies the effects of actions using only domain constraints is too weak and cannot deal with the well known problems such as the frame, qualification, and ramification problems. The reference [3] adresses this problems and intoduces a ternary predicate *Caused* to situation calculus to solve them. In this research the formalism introduced by [12] will be used. According to this formalism, one does not need to know the cause of facts in order to determine whether a world history is causally possible. It is sufficient to know the conditions under which the facts are caused. The causal theory is a set of causal rules which are expressions of the form

$$\phi \Rightarrow \psi \tag{3}$$

where ϕ and ψ are formulas of the underlying propositional language. The intended reading of (3) is: *Necessarily, if ϕ then the fact that ψ is caused.*

It is possible to formalize action domains using this nonmonotonic causal theory. To formalise action domains three pair-wise disjoint sets are defined: a set of *action names*, a set of *fluent names*, and a set of *time instances*. The atoms of the language are expressions of the forms a_t and f_t where *a, f, t* are action, fluent and time instances respectively. a_t means action occurs at time *t* and f_t means fluent *f* holds at time *t*. In this formalism, the causes of the actions are not taken into account and the following two schemas provide that the occurrence of actions is exogenous[2] to the causal theory:

$$a_t \Rightarrow a_t \tag{4}$$

$$f_t \Rightarrow f_t \tag{5}$$

Also the initial values of the fluents are exogenous to the causal theory and this can be represented using the following schemas:

$$f_0 \Rightarrow f_0 \tag{6}$$

$$\neg f_0 \Rightarrow \neg f_0 \tag{7}$$

[1]An implication of the form $\neg q \leftarrow \neg p$ is the contrapositive of $p \leftarrow q$. In a noncontrapositive theory these two statements don't have the same meaning.

[2]An action or fluent is *exogenous* if its cause lies outside of the theory.

Inertia, that is a fluent doesn't change its value unless there is a reason for it, can be expressed using the schema below, where σ is a meta-variable for inertial fluent.

$$\sigma_t \wedge \sigma_{t+1} \Rightarrow \sigma_{t+1} \tag{8}$$

also be described and things that change by themselves can be modeled.

One of the ways of expressing the non-monoticity and negation as failure in logic programming is the "model theoretic" approach. In this approach, the models of the program represent the semantics of the program. Instead of doing query evaluation directly, the intended meaning of the program is obtained by finding its model. As an example to the "model theoretic" approach the well-founded semantics, perfect model semantics, and stable model semantics can be given. Stable model semantics is related with auto-epistemic logic. The stable model of a system is said to be the rational beliefs of an agent. Every predicate in a stable-model is "supported", that is, have a reason to be there. The Smodels system [6] is an implementation of the stable model semantics for range-restricted function-free normal programs. It has two modules: one for finding the stable model of a grounded program and one for grounding the given range-restricted program. There are different front-ends of the Smodels but the richest of them in terms of features is lparse [15].

II PLANNING WITH STABLE MODEL SEMANTICS

The aim of this research is to use logic programming and stable model semantics for plan generation and add it as an option to the CCALC system. The given domain description in *C* will be translated to a logic program and the logic program will be grounded to obtain a ground logic program. The Smodels system accepts ground logic programs and finds their corresponding stable model. The stable model of the program will then be used to find the sought plan. There are different ways of doing all these; one can either take the input written in *C* and directly translate it to a logic program or let CCALC translate it to causal theory and then translate this causal theory to a logic program. In our research, the latter alternative is chosen. Moreover, there are two methods for grounding: let CCALC do grounding and get the ground causal theory or let lparse do grounding, which is the front-end parser of Smodels. In our research the latter method will be applied, because of efficiency consideration.
As a result, the execution steps will be in the following order

- Get the causal theory of the planning problem from CCALC
- Translate it to an equivalent logic program
- Call lparse to ground the program
- Call Smodels to get the answer set (stable model) of the program
- Extract the plan(s) from the stable model of the program

The translation task starts with getting the causal theory from CCALC system. During the translation, instead of using a different predicate for each action and fluent, two predicates will be used. These are *'holds'* and *'occurs'*. The *'holds'* predicate is in the form

$$holds(F, T) \tag{9}$$

meaning that fluent F holds at time T. The *'occurs'* predicate is in the form

$$occurs(A, T) \tag{10}$$

meaning that action A occurs at time T.
One problem is that the early versions of Smodels doesn't have classical negation, but this problem can be easily solved, as suggested in [17], by introducing new predicates standing for the negations of *holds* and *occurs*. The following four rules are used for handling the classical negation problem:

```
 notholds(F,0) :- not holds(F,0), fluent(F).
 notoccurs(A,0) :- not occurs(A,0), action(A).
 holds(F,0) :- not notholds(F,0), fluent(F).
 occurs(A,0) :- not notoccurs(A,0), action(A).
 :- not holds(F,T), not notholds(F,T), fluent(F), time(T), T>0.
 :- not occurs(A,S), not notoccurs(A,S), action(A), step(S), S>0.
```

The other rules of the logic program are obtained by translating the causal theory produced by CCALC. As an example, a solution to the Yale Shooting Problem(YSP) [15] will now be revisited[3].
The *constants* and *variables* are declared as follows[4]:

```
gun(g1).
gun(g2).

action(load(g1)).
action(load(g2)).
action(shoot(g1)).
action(shoot(g2)).
```

Time is represented using the predicates `time` `step` and `next`:

```
time(0).  time(1). time(2).
step(0).  step(1).
next(0,1). next(1,2).
```

`step` is used to make sure that no action takes place at the last time point.
All of the causal rules returned by CCALC are translated to the corresponding logic programming rules. For example, the rule:

```
 shoot(G) causes -alive if loaded(G).
```

becomes

```
 -h(alive,1) <- h(loaded(G),0) && o(shoot(G),0) && true
```

which in turn translated to:

```
notholds(alive,T1) :- holds(loaded(G),T),
      occurs(shoot(G),T),
      time(T),
      next(T,T1),
      gun(G).
```

[3]This example is for giving a complete overview of the translation task. More detailed information about the translation will be given in section 3 and 4

[4]Some of the declarations are omitted for the sake of simplicity.

The initial state and the goal state of the planning domain is written in the *compute* statement of Smodels. For example, for YSP the initial condition is that both of the guns are unloaded and the turkey is alive and the goal to be reached is that the turkey is dead at time 2. So the corresponding *compute* statement is:

```
compute all{ holds(unloaded(g1), 0), holds(unloaded(g2),0),
     holds(alive,0), notholds(alive,2)
   }.
```

The initial state is:

```
holds(unloaded(g1), 0) holds(unloaded(g2),0) holds(alive,0)
```

The goal state is:

```
notholds(alive,2)
```

III TRANSLATION OF CAUSAL RULES TO LOGIC PROGRAM

Causal rules returned by CCALC are in the form:

$$head \Leftarrow body \tag{11}$$

where `head` and `body` can be any kind of logical formulas constructed with the constructs &&, ++, $\rightarrow$, $\leftrightarrow$, $\vee$, $\wedge$. Here && stands for "and", ++ for "or", $\rightarrow$ for "only if", $\leftrightarrow$ for "if and only if", $\vee$ for "there exist" and $\wedge$ for "for all".
Translation starts first by eliminating the constructs $\vee$ and $\wedge$ if there is any. For each expression in the form

$$\vee X F(X) \tag{12}$$

where *F(X)* is an expression having the free variable *X*. The instantiation of *X* results in the formula :

$$F(a_1) + +F(a_2)... + +F(a_n) \tag{13}$$

where a_i is a constant in the domain of the variable *X*. The same method applies to the construct $\wedge$, but in this case ++ is replaced with &&.
The next step is placing this formula in the 'Disjunctive Normal Form' (DNF). The logic program statements are constructed as :

$$\begin{array}{l} head : -b_1 \\ head : -b_2 \\ . \\ . \\ head : -b_n \end{array} \tag{14}$$

where b_i represents a disjunct of the expression that was derived from the original expression by eliminating the constructs $\vee$ and $\wedge$. To illustrate this translation an example from the "gripper problem" can be given. In the CCALC formalization of the gripper problem in [11], the below rule written in *C* language says that: If the gripper of the robot is holding a ball it cannot pick up another ball. The rule is in the form :

```
nonexecutable pickUp(B,G) if \/B1:isHolding(B1,G).
```

The corresponding causal rule returned by CCALC is in the form:

```
(false<-(o(pickUp(B,G),0)&& \/B1:h(isHolding(B1,G),0))&&true)
```

Elimination of $\vee$ results in the expression:

```
 o(pickUp(B,G),0) && ((h(isHolding(ball1,G),0)) \/
                     (h(isHolding(ball2,G),0)) \/
                     (h(isHolding(ball3,G),0)))
```

Placing the expression in DNF results in the expression:

```
 (o(pickUp(B,G),0) && (h(isHolding(ball1,G),0))) \/
 (o(pickUp(B,G),0) && (h(isHolding(ball2,G),0))) \/
 (o(pickUp(B,G),0) && (h(isHolding(ball3,G),0))) \/
```

The corresponding logic program statements are given below:

```
 false :- o(pickUp(B,G),0), (h(isHolding(ball1,G),0))
 false :- o(pickUp(B,G),0), (h(isHolding(ball2,G),0))
 false :- o(pickUp(B,G),0), (h(isHolding(ball3,G),0))
```

IV FORMALIZING PLANNING PROBLEMS AS LOGIC PROGRAMS

CCALC takes a planning problem as a set of facts and goal states. For each fact, the time instance at which it holds, and for each goal state, the time instance at which it should hold is given. On the other hand, the *compute* statement of Smodels system doesn't make any distinction between facts and goals. Writing a literal in the *compute* statement means that the model of the system should include this literal. Therefore the literals in the *compute* statement represent a conjunction. *Compute* statement should include only ground literals.
One of the planning problems for the "gripper problem" given as input to CCALC in [11] is as follows:[5]

```
:- plan
label ::
  0;
facts ::
  0: /\B: at(B,room1),
  0: /\B: color(B,white),
  0: /\B: onFloor(B),
  0: at(robot,room3);
goals ::
  15: (\/B: color(B,red) && \/B: color(B,white) && \/B: color(B,blue)).
```

As in the previous case the translation starts with elimination of the constructs $\vee$ and $\wedge$ if there is any. Then for each fact and goal its corresponding 'Conjunctive Normal Form'(CNF) is found. For example for the goal:

```
  15: (\/B: color(B,red) && \/B: color(B,white) && \/B: color(B,blue)).
```

[5]This is a simplification of the planning problem in [11].

the conjuncts are:

```
  h(at(ball1,white),14) \/ h(at(ball2,white),14) \/ h(at(ball3,white),14)
  h(at(ball1,red),14) \/ h(at(ball2,red),14) \/ h(at(ball3,red),14)
  h(at(ball1,blue),14) \/ h(at(ball2,blue),14) \/ h(at(ball3,blue),14)
```

If the conjunct consists of only one literal it can be directly put in the *compute* statement, as in the planning problem in section 2. Otherwise, a new rule with the same head is created for each literal in the conjunct. The case for the above goal statement is illustrated below and the corresponding *compute* statement is given:

```
temp1:- h(color(ball1,white),14).
temp1:- h(color(ball2,white),14).
temp1:- h(color(ball3,white),14).

temp2:- h(color(ball1,blue),14).
temp2:- h(color(ball2,blue),14).
temp2:- h(color(ball3,blue),14).

temp3:- h(color(ball1,red),14).
temp3:- h(color(ball2,red),14).
temp3:- h(color(ball3,red),14).

compute 1{
h(at(ball1,room1),0), h(at(ball2,room1),0),
h(at(ball3,room1),0), h(color(ball1,white),0),
h(color(ball2,white),0), h(color(ball3,white),0),
h(onFloor(ball1),0), h(onFloor(ball2),0),
h(onFloor(ball3),0), h(at(robot,room3),0),
temp1, temp2, temp3
}.
```

V AN EXAMPLE

Applying the method described in the previous section to YSP, the following logic program is obtained:

```
% Yale Shooting Problem with two guns
gun(g1).
gun(g2).

action(load(g1)).
action(load(g2)).
action(shoot(g1)).
action(shoot(g2)).

inertialFluent(alive).
inertialFluent(loaded(g1)).
inertialFluent(loaded(g2)).

inertialFalseFluent(alive).
inertialFalseFluent(loaded(g1)).
inertialFalseFluent(loaded(g2)).

inertialTrueFluent(alive).
inertialTrueFluent(loaded(g1)).
inertialTrueFluent(loaded(g2)).

fluent(alive).
fluent(loaded(g1)).
fluent(loaded(g2)).

hide.
show o(A,T).
show h(F,T).
show noth(F,T).

time(0).  time(1). time(2).
step(0). step(1).
next(0,1). next(1,2).

% Only one action can be executed at a time
1{o(A,S): action(A)}1 :- step(S).
:- noth(F,T), h(F,T), fluent(F), time(T).

h(F,0) :- h(F,0),fluent(F).
o(A,S) :- o(A,S), action(A), step(S).
noth(F,0) :- noth(F,0), fluent(F).
noto(A,S) :- noto(A,S), action(A), step(S).

% Rules for implementing the classical negation
noth(F,0) :- not h(F,0), fluent(F).
noto(A,S) :- not o(A,S),  action(A), step(S).

h(F,0) :- not noth(F,0), fluent(F).
o(A,S) :- not noto(A,S),  action(A), step(S).

:- not h(F,T), not noth(F,T), fluent(F), time(T), T>0.
:- not o(A,S), not noto(A,S), action(A), step(S), S>0.

% The following rules are written by translating the causal theory
% returned by CCALC.

h(loaded(G),T1) :- o(load(G),S), step(S), time(T1),next(S,T1),
   gun(G).
```

```
noth(alive,T1) :- h(loaded(G),S), o(shoot(G),S), step(S),time(T1),
                  next(S,T1), gun(G).

noth(loaded(G),T1) :-o(shoot(G),S), step(S), time(T1), next(S,T1),
     gun(G).

h(IT,T) :- h(IT,S), not noth(IT,T), step(S), time(T), next(S,T),
           inertialTrueFluent(IT).

h(DT,T) :- h(DT,T), defaultTrueFluent(DT), time(T), T>0.

noth(DF,T) :- noth(DF,T), defaultFalseFluent(DF), time(T), T>0.

noth(IF,T) :- noth(IF,S), not h(IF,T), step(S), time(T), next(S,T),
              inertialFalseFluent(IF).

% The initial and final states
compute all{
  h(alive,0), noth(loaded(g1),0), noth(loaded(g2),0), noth(alive,2)
}.
```

The Smodels system finds two stable models for this program. These are:

```
Answer: 1
Stable Model: noth(loaded(g1),0) noth(loaded(g2),0) h(alive,0)
noth(alive,2) o(shoot(g2),1) h(loaded(g2),1) noth(loaded(g2),2)
noth(loaded(g1),2) noth(loaded(g1),1) h(alive,1) o(load(g2),0)
Answer: 2
Stable Model: noth(loaded(g1),0) noth(loaded(g2),0) h(alive,0)
noth(alive,2) o(shoot(g1),1) h(loaded(g1),1) noth(loaded(g2),2)
noth(loaded(g2),1) noth(loaded(g1),2) h(alive,1) o(load(g1),0)
```

The first model gives a plan in which the gun `g2` is loaded and shooted and the second model gives a plan in which the other gun `g1` is loaded and shooted.

VI CONCLUSION

In this research, the aim is to use the action description language *C* to define actions and their effects on fluents. By translating *C* program to a logic program, it will be possible to combine the expressiveness of *C* with efficient computational techniques for logic programming. Finding the stable model of the logic program using Smodels will provide the required planning for a given domain.

REFERENCES

[1] Chitta Baral and Michael Gelfond. "Logic programming and knowledge representation." *Journal of Logic Programming*, 19(20), pp. 74-148, (1994).

[2] Enrico Giunchiglia and Vladimir Lifschitz. “An action language based on causal explanation: Preliminary report.” In *Proc. AAAI-98*, pp. 623-630, (1998).

[3] Fangzhen Lin. “Embracing causality in specifying the indirect effects of actions.” In *Proc. of IJCAI-95, 1985-1991* .

[4] Hanto Zhang. “SATO: An efficient propositional prover.” In *Proc. of International Conference of Automated Deduction (CADE-97), (1997).*

[5] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI-92*, pp. 359-379, (1992).

[6] Ilkka Niemelä and Patrik Simons. “Efficient implementation of the well-founded and stable model semantics.” In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pp. 289-303, (1996).

[7] Ilkka Niemelä, Patrik Simons and Timo Soininen. Stable model semantics and weight constraint rules. In *Proc. of the Fifth International Conference on Logic Programming and Non-Monotonic Reasoning*, (1999).

[8] John McCarthy. “Elaboration tolerance.” In progress, 1999. [Available at `http://www-formal.stanford.edu/jmc/elaboration.html`]

[9] Joohung Lee and Vladimir Lifschitz.Additive fluents(DRAFT),2000. [Available at `http://www.cs.utexas.edu/users/vl/mypapers/additive.ps`]

[10] Michael Gelfond and Vladimir Lifschitz. “Action languages.” *Electronic Transaction on AI* 3, (1998).

[11] Norman McCain. Using the Causal Calculator(Draft),1999. [Available at `http://www.cs.utexas.edu/users/mccain/cc/ccalc/manual-C.ps`]

[12] Norman McCain and Hudson Turner. “Causal Theories of action and change.” In *Proc. AAAI-97*, pp. 460-465, (1997).

[13] Roberto Bayardo and Robert Schrag. “Using the CSP look-back techniques to solve real-world SAT instances.” *In Proc. of AAAI-97*, pp. 203-208, (1997).

[14] Patrik Simons. “Extending the stable model semantics with more expressive rules.” In *Proc. of the 5th International Conference on Logic Programming and Non-monotonic Reasoning*, (1999).

[15] Tommi Syrjänen, “Lparse User’s Manual (Draft 0.9)”,1999.[Available at `http://www.tcs.hut.fi/Software/smodels/lparse/lparse.ps.gz`]

[16] Vladimir Lifschitz. “Missionaries and Cannibals in the Causal Calculator.” In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int’l Conf.*, pp. 85-96, (2000).

[17] Vladimir Lifschitz. “Action languages, answer sets and planning.” In *The Logic Programming Paradigm: A 25-Year Perspective*, Springer-Verlag, (1999).