Additive Fluents

Joohyung Lee and Vladimir Lifschitz
Department of Computer Sciences
University of Texas
Austin, TX 78712
{appsmurf,vl}@cs.utexas.edu

January 1, 2001

$* * *$ DRAFT $* * *$

Abstract

An "additive" fluent is a fluent with numerical values such that the effect of several concurrently executed actions on it can be computed by adding the effects of the individual actions. We show how queries about additive fluents can be answered using the Causal Calculator. The examples discussed in this paper involve buying and selling, applying forces and moving groups of objects. The additive fluent mechanism can be used to instruct the Causal Calculator to generate more efficient plans.

1 Introduction

In the theory of knowledge representation, the term "fluent" [McCarthy and Hayes, 1969] refers to a function defined on the space of situations, that is, to anything that depends on the state of the world. For instance, the sentence *It is raining in Austin* represents a "propositional fluent," or a function from situations to truth values: in some situations this sentence is true, in others false. *The number of full-time students at the University of Texas* is a fluent whose values are integers.

Most authors who write on representing properties of actions limit their attention to propositional fluents. In a description of the blocks world, for instance, the assertion that block $B1$ is on top of block $B2$ would be usually written as $on(B1, B2)$ rather than $loc(B1) = B2$. Geffner [2000] shows how to incorporate nonpropositional fluents in the language of STRIPS.

In this paper we identify a special kind of fluents with numerical values—we call them "additive"—and show how reasoning about additive fluents can be automated using the Causal Calculator.[1]

What distinguishes additive fluents is that the effect of several concurrently executed actions on such a fluent can be computed by adding the effects of the individual actions. For instance, when 3 new students enroll at the university on the same day, the total number of students is incremented by the sum $1 + 1 + 1$ of the "contributions" of their actions. Graduating from the university can be thought of as an action that contributes -1 to the total number of students; accordingly, when m students enroll and n students graduate, the net contribution of this set of actions equals $m - n$.

Another example of an additive fluent is given by a bank account balance: the net effect of several deposits and withdrawals equals the sum of the effects of the individual transactions. Similarly, the amount of liquid in a container is an additive fluent. The voltage of a battery is an additive fluent, in the sense that the increase in the voltage caused by adding several cells to a battery can be computed by addition. In mechanics, the velocity of a particle is an additive fluent, because the net effect of several forces on this fluent over a time interval equals the sum of the effects of the individual forces. Additive fluents are ubiquitous, which is one of the reasons why adding numbers is such a useful operation.

Our interest in this class of fluents was originally motivated by a difficulty encountered in formalizing some elaborations of the Missionaries and Cannibals puzzle in [Lifschitz, 2000]. One of the postulates adopted in that paper is that if the number of members of a group (say, missionaries) in some location (say, the left bank of the river) equals x, and a vessel arrives with y members of the group aboard, the number will become $x + y$. But this may be incorrect when several actions are executed concurrently. If, for instance, a boat is taking y missionaries to the left bank while another boat is taking z missionaries to the right bank then the number will become $x + y - z$. To treat such examples correctly, we need to view the number of members of a group in a location as an additive fluent.

Fluents in the examples above are what we call "additive-inertial." The new value of such a fluent is obtained by adding the contributions of all concurrently executed actions to its old value. In particular, if an additive-inertial fluent is not affected by any of the actions that have just been executed then its new value equals its old value, in the spirit of the commonsense law of inertia. We will introduce also "additive-default-zero"

[1]This system can be downloaded from `http://www.cs.utexas.edu/users/tag/cc` . The experiments described in this draft use a new version of the Causal Calculator that will be released later this year.

fluents; the new value of such a fluent *equals* the sum of the contributions of all concurrently executed actions. If a fluent of this kind is not affected by any of the actions that have just been executed then its new value is 0. The number of students who enrolled at the university *today* is an additive-default-zero fluent, and so is the total amount withdrawn *today* from a bank account.

In literature on planning, fluents with numerical values are often referred to as "resources" [Koehler, 1998]. The concurrent execution of the actions that involve resources is often limited to the case when all ways of sequencing the concurrent actions are well-defined and equivalent. The formalization of this condition in Section 4.1 of [Kautz and Walser, 1999] involves an equation that is similar to the characterization of additive-inertial fluents given above.

After a brief discussion of the Causal Calculator (CCALC) in Section 2, we give several examples of its use that involve additive fluents. The first example has to do with buying and selling (Section 3); the second, with classical mechanics (Section 4); the third is a missionaries and cannibals problem with two boats (Section 5). In Section 6 we show how the additive fluent mechanism can be used to instruct CCALC to generate more economical plans. The Appendix contains the listing of the standard include file `additive` which describes additive fluents with integer values in the input language of CCALC.

2 Causal Calculator

The Causal Calculator is an implementation of the propositional causal logic from [McCain and Turner, 1997], written in Prolog. A part of its input language serves for describing transition systems[2] and is based on action language $\mathcal{C}$ from [Giunchiglia and Lifschitz, 1998]. A description of a transition system is translated by CCALC first into a causal theory and then into a set of propositional formulas. The models of this set of formulas correspond to paths in the given transition system. Satisfiability solvers, such as SATO [Zhang, 1997] and RELSAT [Bayardo and Schrag, 1997], are used for search, in the spirit of satisfiability planning [Kautz and Selman, 1992]. Since causal logic is closely related to logic programming under the answer set semantics [Gelfond and Lifschitz, 1991], the use of CCALC can be viewed as a form of answer set programming [Marek and Truszczyński, 1999], [Niemelä, 1999], [Lifschitz, 1999]. Examples of the use of CCALC as

[2]A transition system represents an action domain by a directed graph whose vertices correspond to states, and whose edges correspond to the execution of actions [Lifschitz, 1997].

a planner can be found in [McCain and Turner, 1998], [Babovich *et al.*, 2000], [Erdem *et al.*, 2000], [Lifschitz, 2000] and [Lifschitz *et al.*, 2000].

As a simple example, consider the description of the blocks world in the language of CCALC shown in Figure 1. The file begins with declaring a sort `location`, its subsort `block`, and several variables of these sorts. Then object constant `table` of sort `location` is declared. Function constant `loc` represents an operation that turns a block into a fluent. This fluent is inertial (that is, it tends to keep the value that it had in the past), and the values of this fluent are locations. The operation denoted by `move` turns a block into an action—moving that block. Operation `to` gives an attribute of that action, which is a location—the destination of the move.

The constant declarations are followed by a proposition that describes the effect of moving a block: moving `B` causes `loc(B)` to be equal to `L` if `L` is the destination of the move. Symbols `causes` and `if` come from the syntax of action language $\mathcal{C}$, and they are reserved words in the language of CCALC. Infix operator `eq` is a reserved word also; its first argument represents a fluent, and the second argument represents the current value of that fluent. The next three propositions are constraints on the executability of actions; `nonexecutable` is a reserved word, and `&&` is the symbol for conjunction. The last proposition expresses a constraint on possible states of the world; `always` is a reserved word, and `->>` is the symbol for material implication.

Figure 2 shows how this description of the blocks world can be used to solve a planning problem. The `plan` directive specifies the initial conditions and the goal of the problem; symbols `0:` and `2:` are "time stamps." The `show` directive instructs CCALC to display the locations of blocks at every step, in addition to the plan that it generates.

Given this input file, CCALC responds:

```
calling sato 3.1.2...
run time (seconds)                0.01

0:  loc(a) eq b  loc(b) eq table  loc(c) eq d  loc(d) eq table

ACTIONS:  move(a,to:table)  move(c,to:table)

1:  loc(a) eq table  loc(b) eq table  loc(c) eq table
loc(d) eq table

ACTIONS:  move(b,to:a)  move(d,to:c)

2:  loc(a) eq table  loc(b) eq a  loc(c) eq table  loc(d) eq c
```

Further examples of CCALC input files in this paper use a few

```
:- sorts
  location >> block.

:- variables
  B,B1,B2                  :: block;
  L                        :: location.

:- constants
  table                    :: location;
  loc(block)               :: inertialFluent(location);
  move(block)              :: action;
  to(block)                :: attribute(move,location).

% effect of moving a block
move(B) causes loc(B) eq L if to(B) eq L.

% a block can be moved only when it is clear
nonexecutable move(B) if loc(B1) eq B.

% a block can be moved only to a position that is clear
nonexecutable move(B) if to(B) eq B1 && loc(B2) eq B1.

% a block can't be moved onto a block that is being moved also
nonexecutable move(B) && move(B1) if to(B) eq B1.

% two blocks can't be on the same block at the same time
always loc(B1) eq B && loc(B2) eq B ->> B1=B2.
```

Figure 1: File `bw.t`: The blocks world

```
% Find a 2-step plan:
%
% initial condition       goal
%
%    a   c                b   d
%    b   d                a   c
%  ---------            ---------

:- include 'bw.t'.

:- constants
  a,b,c,d         :: block.

:- plan
facts::
  0: loc(a) eq b,
  0: loc(b) eq table,
  0: loc(c) eq d,
  0: loc(d) eq table;
goals::
  2: loc(a) eq table,
  2: loc(b) eq a,
  2: loc(c) eq table,
  2: loc(d) eq c.

:- show loc(B) eq L.
```

Figure 2: File `bw-test.t`: A blocks world planning problem

```
:- sorts
  integer >> nnInteger.

:- constants
  minInt..maxInt      :: integer;
  0..maxInt           :: nnInteger.

:- macros
  sum(#1,#2,#3) ->
        #1 is max(minInt, min((#2)+(#3),maxInt));
  diff(#1,#2,#3) ->
        #1 is max(minInt, min((#2)-(#3),maxInt));
  prod(#1,#2,#3) ->
        #1 is max(minInt, min((#2)*(#3),maxInt)).
```

Figure 3: Standard file `arithmetic`

symbols related to integer arithmetic: sort `integer`, which covers the numbers between `minInt` and `maxInt`; its subsort `nnInteger`, covering the nonnegative numbers in that interval; symbols `sum`, `diff` and `prod` representing operations on the integers between `minInt` and `maxInt`. The choice of the values of these two constants depends on the specific problem that CCALC is expected to solve. The other symbols mentioned above are defined in standard file `arithmetic` (Figure 3). Symbols `#1`, `#2`, `#3` in that file serve as parameters in the CCALC macro expansion mechanism.

3 Buying and Selling

Figure 4 describes properties of buying and selling. The standard file `additive`, included in this description, is reproduced in the appendix. It formalizes the idea of an integer-valued additive fluent in the language of CCALC, and includes, in particular, file `arithmetic` shown in Figure 3.

Expressions of the form `have(agent,resource)` are declared to be additive-inertial fluents with nonnegative values. Recall that we think of the effect of an action on an additive fluent in terms of that action's contribution; the effect of a set of concurrently executed actions is computed as the sum of the contributions of the individual actions. Accordingly, we describe the effects of actions on the fluents `have(agent,resource)` using the reserved words `increments` and `decrements`, rather than `causes` used in Figure 1. For instance, the last proposition in Figure 4 says that buying decrements the amount of money that the buyer has by the price of the item times the number of items bought.

As an example of the use of this little theory, we can ask CCALC:

> I have \$6 in my pocket. A newspaper costs \$1, and a magazine costs \$3. Do I have enough money to buy 2 newspapers and a magazine? A newspaper and 2 magazines?

In Figure 5, each of these questions is interpreted as the question of the executability of a plan that consists of two concurrent actions. The file begins with defining the boundaries of sort `integer` to be -7 and 7. In the `plan` query with `label` 1, we assume that the action `buy(newspaper)` occurs with the `howmany` attribute 2, and the action `buy(magazine)` occurs with the `howmany` attribute 1. In the second query, the values of the attributes are reversed.

In response to the first query, CCALC says:

```
calling sato 3.1.2...
run time (seconds)                 0.23

0:  have(buyer,money) eq 6

ACTIONS:  buy(magazine,howmany:1)  buy(newspaper,howmany:2)

1:  have(buyer,money) eq 1
```

Its reply to the second query is

```
calling sato 3.1.2...
run time (seconds)                 0.22
  No plan of length 1
```

Note that the initial conditions in this example are incomplete. We are given the initial values of fluents `price(newspaper)`, `price(magazine)` and `have(buyer,money)`, but we are not told how much money the seller has, and we are not given the number of newspapers or magazines initially available on either side. The positive answer given by CCALC to the first query means only that the given actions are executable in *at least one* state satisfying the initial conditions. There exist valid initial states in which the given actions are not executable: no matter how much money

```
:- include 'additive'.

:- sorts
  agent;
  resource >> item.

:- variables
  Ag                          :: agent;
  Res                         :: resource;
  It                          :: item;
  M,N                         :: nnInteger;
  X                           :: computed.

:- constants
  buyer,seller                :: agent;
  money                       :: resource;
  price(item)                 :: inertialFluent(nnInteger);
  have(agent,resource)        :: nnAdditiveIFluent;
  buy(item)                   :: action;
  howmany(item)               :: attribute(buy,nnInteger).

buy(It) increments have(buyer,It) by N
   if howmany(It) eq N.

buy(It) decrements have(seller,It) by N
   if howmany(It) eq N.

buy(It) increments have(seller,money) by X
   if price(It) eq M && howmany(It) eq N && prod(X,M,N).

buy(It) decrements have(buyer,money) by X
   if price(It) eq M && howmany(It) eq N && prod(X,M,N).
```

Figure 4: File `buying.t`: Buying and selling

```
:- macros
  minInt -> -7;
  maxInt -> 7.

:- include 'buying.t'.

:- constants
  newspaper,magazine          :: item.

:- plan
label :: 1;
facts ::
 0: price(newspaper) eq 1,
 0: price(magazine) eq 3,
 0: have(buyer,money) eq 6,
 0: o(buy(newspaper)),
 0: howmany(newspaper) eq 2,
 0: o(buy(magazine)),
 0: howmany(magazine) eq 1.

:- plan
label :: 2;
facts ::
 0: price(newspaper) eq 1,
 0: price(magazine) eq 3,
 0: have(buyer,money) eq 6,
 0: o(buy(newspaper)),
 0: howmany(newspaper) eq 1,
 0: o(buy(magazine)),
 0: howmany(magazine) eq 2.

:- show have(buyer,money) eq N.
```

Figure 5: File `buying-test.t`: Do I have enough cash?

the buyer has, he can't buy a newspaper if the seller doesn't have one available. This "weak" understanding of executability seems to be an adequate representation of the question *Do I have enough money?*.

4 Space Travel

Some additive fluents mentioned in the introduction—for instance, the velocity of a particle—are real-valued, rather than integer-valued. Since file `additive` does not deal with real numbers, it does not allow us to formalize properties of such fluents.

But let's imagine a movable object that is immune to this complication—the spacecraft Integer. Far away from stars and planets, the Integer is not affected by any external forces. As its proud name suggests, the mass of the spacecraft is an integer. For every integer t, the coordinates and all three components of the Integer's velocity vector at time t are integers; the forces applied to the spacecraft by its jet engines over the interval $(t, t+1)$, for any integer t, are constant vectors whose components are integers as well. If the crew of the Integer attempts to violate any of these conditions, the jets will fail to operate!

The motion of the Integer is described in Figure 6. The three fluents of the form `pos(axis)` represent the current position of the Integer. The additive fluents `vel(axis)` are the components of its velocity. According to Newton's Second Law, the acceleration created by firing a jet can be computed by dividing the force by the mass of the spacecraft. The first proposition in Figure 6 expresses this fact without mentioning the acceleration explicitly, in terms of the change in the velocity over a unit time interval. Symbol `//` stands for integer division; the second proposition tells us that firing a jet is impossible if this division gives a nonzero remainder.

The third proposition says that the position of the spacecraft at time $t + 1$ can be computed by adding its average velocity over the interval $(t, t+1)$ to its position at time t. Because the acceleration over this interval is constant, the average velocity is computed as the arithmetic mean of the velocities at times t and $t + 1$. We do not include any assumptions about the case when the division by 2 involved in computing this arithmetic mean produces a fraction. Some features of the semantics of CCALC that we will not discuss here guarantee actually that firing jets to achieve this result would be impossible.

Finally, to make planning for the Integer more interesting, we use the constant `maxForce` to limit the power of the jets.

To test this representation, we ask the following question in Figure 7:

```
:- include 'additive'.

:- sorts
  axis; jet.

:- variables
  Ax                                :: axis;
  J                                 :: jet;
  F,V,V1,P                          :: integer;
  X,Y,Z                             :: computed.

:- constants
  x,y,z                             :: axis;
  jet1,jet2                         :: jet;
  pos(axis)                         :: fluent(integer);
  vel(axis)                         :: additiveIFluent;
  fire(jet)                         :: action;
  force(jet,axis)                   :: attribute(fire,integer).

fire(J) increments vel(Ax) by X
   if force(J,Ax) eq F && X is F // mass.

nonexecutable fire(J)
   if force(J,Ax) eq F && F mod mass =\= 0.

caused pos(Ax) eq X if vel(Ax) eq V1
   after vel(Ax) eq V && pos(Ax) eq P && sum(Y,V,V1)
      && Z is Y//2 && Y mod 2 =:= 0 && sum(X,P,Z).

nonexecutable fire(J)
   if force(J,Ax) eq F && abs(F)>maxForce.
```

Figure 6: File `spacecraft.t`: The spacecraft Integer

```
:- macros
  mass -> 1;
  maxForce -> 2;
  minInt -> -7;
  maxInt -> 7.

:- include 'spacecraft.t'.

:- plan
facts ::
 0: (pos(x) eq -1  &&  pos(y) eq  0  &&  pos(z) eq  1),
 0: (vel(x) eq  0  &&  vel(y) eq  1  &&  vel(z) eq  1);
goals ::
 1: (pos(x) eq  0  &&  pos(y) eq  3  &&  pos(z) eq  1).

:- show pos(Ax) eq P; vel(Ax) eq P.
```

Figure 7: File `spacecraft-test.t`: How to get there?

The position of the Integer is $(-1, 0, 1)$, and its velocity is $(0, 1, 1)$. How can it get to $(0, 3, 1)$?

Here is the output produced by CCALC:

```
calling sato 3.1.2...
run time (seconds)                0.34

0:  vel(x) eq 0  vel(y) eq 1  vel(z) eq 1  pos(x) eq -1
pos(y) eq 0  pos(z) eq 1

ACTIONS:  fire(jet1,force(x):2,force(y):2,force(z):0)
fire(jet2,force(x):0,force(y):2,force(z): -2)

1:  vel(x) eq 2  vel(y) eq 5  vel(z) eq -1  pos(x) eq 0
pos(y) eq 3  pos(z) eq 1
```

5 Missionaries and Cannibals

In the Missionaries and Cannibals Problem (MCP), three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross? The shortest solution is known to involve 11 steps.

Lifschitz [2000] showed how to express MCP in the language of CCALC. The representation used in that paper does not introduce names for individual missionaries or cannibals; rather, a state is described in terms of the number of members of each group on each bank of the river.

In the modification of MCP discussed in this section, the travelers find two boats: a small boat that holds one, and a bigger boat that holds two.[3] Is the modified problem solvable in fewer than 11 steps?

This modification is more difficult to formalize than the original form of MCP. As discussed in the introduction, the number of members of a group `G` in a location `L` should be treated as an additive fluent when several boats are available. Accordingly, in the formalization of the modified MCP shown in Figures 8–10, fluents `num(G,L)` are described as additive-inertial fluents. And there is one more difficulty. Imagine that there is a single cannibal on the left bank, with a boat that holds two. In this state, our postulates should make it impossible, of course, for *two* cannibals to cross in that boat. In the absence of other boats, we didn't have to worry about this case: if there is

[3]This is similar, but not quite equivalent, to Elaboration 10 of MCP from [McCarthy, 1999]: One of the missionaries is Jesus Christ.

only one cannibal on the left bank then two cannibals leaving would have made the new number of cannibals on the left bank negative; consequently, such an action is nonexecutable. With two boats, this reasoning does not apply anymore: a third cannibal, crossing simultaneously in the opposite direction, would make the new number of cannibals on the left bank equal to 0, which is a legal value.

To prohibit such actions, we need to say that the total number of members of a group `G` leaving a location `L` does not exceed the number of members of that group in that location. Our formalization expresses this constraint using the auxiliary fluent `departed(G,L)`: the total number of members of group `G` who just departed from location `L`. This is an additive-default-zero fluent (see the discussion of the two kinds of additive fluents in the introduction).

The constraint at the beginning of Figure 9 says that, in any valid state, the missionaries are not outnumbered by the cannibals in any location. Then we describe the effects of crossing on the location of the vessel, on the number of members of any group `G` at the point of departure, on the number of members of group `G` at destination, and on the total number of members of group `G` who have just departed. Finally, we state four restrictions on the executability of crossing: the destination should be different from the point of departure; there should be at least one person aboard; the total number of people aboard should not exceed the capacity of the vessel; the total number of members of a group leaving any location should not exceed the number of members of that group in that location.

The pair `6..7` of time stamps in Figure 10 says to CCALC: try to solve the problem in 6 steps, and, if you determine that there is no such solution, try 7 steps. The following output is produced:

```
calling sato 3.1.2...
run time (seconds)                73.52
  No plan of length 6,



calling sato 3.1.2...
run time (seconds)                83.76

0:  num(ca,bank1) eq 3  num(ca,bank2) eq 0  num(mi,bank1) eq 3
num(mi,bank2) eq 0  loc(boat1) eq bank1  loc(boat2) eq bank1

ACTIONS:  cross(boat2,to:bank2,howmany(ca):1,howmany(mi):1)
```

```
:- include 'additive'.

:- macros
  outnumbered(#1,#2) -> (#2 > #1) && (#1 > 0).

:- sorts
  vessel;
  location;
  group.

:- variables
  V                                 :: vessel;
  L                                 :: location;
  G                                 :: group;
  M,N,K                             :: nnInteger;
  X                                 :: computed.

:- constants
  loc(vessel)                       :: inertialFluent(location);
  num(group,location)               :: nnAdditiveIFluent;
  departed(group,location)          :: nnAdditiveDZFluent;
  cross(vessel)                     :: action;
  to(vessel)                        :: attribute(cross,location);
  howmany(vessel,group)             :: attribute(cross,nnInteger);
  boat1,boat2                       :: vessel;
  bank1,bank2                       :: location;
  mi,ca                             :: group;
  capacity(vessel)                  :: fluent(nnInteger).

% boat capacities

caused capacity(boat1) eq 1.
caused capacity(boat2) eq 2.
```

Figure 8: File `mcp.t`: Modified MCP, Part 1

```
% constraint

never num(mi,L) eq M && num(ca,L) eq N && outnumbered(M,N).

% effects of crossing

cross(V) causes loc(V) eq L if to(V) eq L.

cross(V) decrements num(G,L) by N
   if loc(V) eq L && howmany(V,G) eq N.

cross(V) increments num(G,L) by N
   if to(V) eq L && howmany(V,G) eq N.

cross(V) increments departed(G,L) by N
   if howmany(V,G) eq N && loc(V) eq L.

% executability of crossing

nonexecutable cross(V) if loc(V) eq L && to(V) eq L.

nonexecutable cross(V) if (/\G: howmany(V,G) eq 0).

nonexecutable cross(V)
   if howmany(V,mi) eq M && howmany(V,ca) eq N
   && sum(X,M,N) && capacity(V) eq K && X>K.

caused false if departed(G,L) eq M
   after num(G,L) eq N && M>N.
```

Figure 9: File `mcp.t`: Modified MCP, Part 2

```
:- macros
  minInt -> -3;
  maxInt -> 3.

:- include 'mcp.t'.

:- plan
facts ::
0: (num(mi,bank1) eq 3 && num(ca,bank1) eq 3),
0: (num(mi,bank2) eq 0 && num(ca,bank2) eq 0),
0: (/\V: loc(V) eq bank1);
goals ::
6..7: (num(mi,bank1) eq 0 && num(ca,bank1) eq 0).

:- show loc(V) eq L; num(G,L) eq N.
```

Figure 10: File `mcp-test.t`: How shall they cross?

```
1:  num(ca,bank1) eq 2  num(ca,bank2) eq 1  num(mi,bank1) eq 2
num(mi,bank2) eq 1  loc(boat1) eq bank1  loc(boat2) eq bank2

ACTIONS:  cross(boat2,to:bank1,howmany(ca):0,howmany(mi):1)

2:  num(ca,bank1) eq 2  num(ca,bank2) eq 1  num(mi,bank1) eq 3
num(mi,bank2) eq 0  loc(boat1) eq bank1  loc(boat2) eq bank1

ACTIONS:  cross(boat1,to:bank2,howmany(ca):0,howmany(mi):1)
cross(boat2,to:bank2,howmany(ca):0,howmany(mi):2)

3:  num(ca,bank1) eq 2  num(ca,bank2) eq 1  num(mi,bank1) eq 0
num(mi,bank2) eq 3  loc(boat1) eq bank2  loc(boat2) eq bank2

ACTIONS:  cross(boat2,to:bank1,howmany(ca):1,howmany(mi):0)

4:  num(ca,bank1) eq 3  num(ca,bank2) eq 0  num(mi,bank1) eq 0
num(mi,bank2) eq 3  loc(boat1) eq bank2  loc(boat2) eq bank1
```

```
ACTIONS:  cross(boat1,to:bank1,howmany(ca):0,howmany(mi):1)
cross(boat2,to:bank2,howmany(ca):2,howmany(mi):0)

5:  num(ca,bank1) eq 1  num(ca,bank2) eq 2  num(mi,bank1) eq 1
num(mi,bank2) eq 2  loc(boat1) eq bank1  loc(boat2) eq bank2

ACTIONS:  cross(boat1,to:bank2,howmany(ca):0,howmany(mi):1)
cross(boat2,to:bank1,howmany(ca):1,howmany(mi):0)

6:  num(ca,bank1) eq 2  num(ca,bank2) eq 1  num(mi,bank1) eq 0
num(mi,bank2) eq 3  loc(boat1) eq bank2  loc(boat2) eq bank1

ACTIONS:  cross(boat2,to:bank2,howmany(ca):2,howmany(mi):0)

7:  num(ca,bank1) eq 0  num(ca,bank2) eq 3  num(mi,bank1) eq 0
num(mi,bank2) eq 3  loc(boat1) eq bank2  loc(boat2) eq bank2
```

6 Improving Plans

In satisfiability planning and in answer set planning, when a plan without concurrent actions is desired, it is usual to make the process of plan generation more efficient by allowing a subset of actions to be executed concurrently as long as that subset is "serializable." In such a plan, the actions that are scheduled for the same time period can be instead executed consecutively, in any order. For example, the restrictions on blocks world plans in file `bw.t` (Figure 1) ensure serializability. The 2-step plan shown in Section 2 can be turned into a 4-step sequential plan by ordering the actions `move(a,to:table)` and `move(c,to:table)` in an arbitrary way, and then ordering `move(b,to:a)` and `move(d,to:c)` in an arbitrary way.

Generating serializable solutions is a computationally useful trick, but there is a difficulty associated with it. As observed in [Kautz and Walser, 1999], when the shortest possible serializable plan is found, we cannot generally expect that a sequential plan obtained from it by serialization will be optimal in the sense of the number of steps. Consider, for instance, the blocks world benchmark problem large.c from [Kautz and Selman, 1996] and [Niemelä, 1999]. The problem involves 15 blocks. The shortest serializable solution to this problem consists of 8 steps. Such a solution, found by CCALC using SATO as the search engine, includes 52 moves; there will be only 38 moves, however, if RELSAT is used instead. The shortest serializable solution to large.c found by SMODELS 2.25 on the basis of the formalization given in [Niemelä, 1999] consists of 29 moves. But all these

```
:- include 'additive'; 'bw.t'.

:- constants
  cost          :: nnAdditiveIFluent.

move(B) increments cost by 1.
```

Figure 11: File `bw-cost.t`: Computing the cost of a plan

numbers are actually much larger than necessary: as discussed below, there exists a serializable solution to large.c that has length 8 and consists of 18 moves.

Let's define the *cost* of a solution to a blocks world planning problem to be the total number of move actions in it. In the case of a serializable plan, this is the same as the length of a sequential plan obtained from it by serialization. Using the additive fluent mechanism, we can easily characterize the cost of a plan in the language of CCALC (Figure 11). Then CCALC can be used to check whether the cost of a plan that it has found is minimal. For instance, in Figure 12 we instruct CCALC to find a serializable solution to large.c whose length is 8 and whose cost is at most 18. It produces the following plan:

```
calling relsat 1.1.2...
Prep time : 248 seconds.  Solve time : 219 seconds.

0:  cost eq 0

ACTIONS:  move(c,to:table)  move(i,to:table)  move(k,to:table)

1:  cost eq 3

ACTIONS:  move(b,to:table)  move(h,to:table)  move(j,to:table)

2:  cost eq 6

ACTIONS:  move(e,to:j)  move(k,to:g)

3:  cost eq 8
```

```
:- macros
  minInt  -> 0;
  maxInt  -> 19;
  length  -> 8;
  maxCost -> 18.

:- include 'bw-cost.t'.

:- variables
  N                               :: nnInteger.

:- constants
  a,b,c,d,e,f,g,h,i,j,k,l,m,n,o  :: block.

:- plan
facts ::
0:      (cost eq 0  &&  loc(m) eq table  &&  loc(l) eq m
        &&  loc(a) eq l  &&  loc(b) eq a  &&  loc(c) eq b
        &&  loc(o) eq table  &&  loc(n) eq o  &&  loc(d) eq n
        &&  loc(e) eq d  &&  loc(j) eq e  &&  loc(k) eq j
        &&  loc(f) eq table  &&  loc(g) eq f  &&  loc(h) eq g
        &&  loc(i) eq h);
goals ::
length: ((\/N: (cost eq N && N=<maxCost))
        &&  loc(e) eq j  &&  loc(a) eq e  &&  loc(n) eq a
        &&  loc(i) eq d  &&  loc(h) eq i  &&  loc(m) eq h
        &&  loc(o) eq m  &&  loc(k) eq g  &&  loc(c) eq k
        &&  loc(b) eq c  &&  loc(l) eq b).

:- show cost eq N.
```

Figure 12: Finding an economical solution to large.c

```
ACTIONS:  move(a,to:e)  move(c,to:k)  move(d,to:table)

4:  cost eq 11

ACTIONS:  move(b,to:c)  move(i,to:d)  move(n,to:a)

5:  cost eq 14

ACTIONS:  move(h,to:i)  move(l,to:b)

6:  cost eq 16

ACTIONS:  move(m,to:h)

7:  cost eq 17

ACTIONS:  move(o,to:m)

8:  cost eq 18
```

If we make `maxCost` equal to 17 then CCALC will tell us that the problem is not solvable.

It is interesting to note that large.c has sequential solutions whose length is less than 18. Such solutions cannot be obtained, however, by serializing short concurrent solutions. This kind of trade-off between the length of a serializable solution and the length of the corresponding sequential solution was demonstrated by Kautz and Walser [1999] in the logistics domain. We have used CCALC to investigate this phenomenon in the case of problem large.c. According to the results shown in Figure 13, the length of the shortest sequential solution is 14, but such a plan cannot be obtained from a serializable plan whose length is less than 13.

7 Conclusion

We have seen that additive fluents play an important role in knowledge representation, and that some examples of reasoning about additive fluents can be automated using the Causal Calculator. The same computational method would be applicable to other answer set solvers. In the context of SMODELS, weight constraints [Simons, 1999] would be a useful representational tool.

Length of serializable plan	Smallest possible cost
8	18
9	16
10	15
11	15
12	15
13	14
14	14

Figure 13: Trade-off between length and cost in solutions to large.c

The current version of CCALC does not operate with real numbers, and even integer arithmetic is implemented in a way that becomes inefficient when large integers are needed. It may be possible to lift these limitations by developing an interface with search engines other than propositional solvers, such as those based on linear programming, as in [Wolfman and Weld, 1999], or on integer programming, as in [Kautz and Walser, 1999].

Acknowledgements

The work described in this paper would have been impossible without Norman McCain, who wrote the Causal Calculator, and Esra Erdem, who added nonpropositional fluents to it and extended it in many other ways. The term "additive" was suggested to us by Hudson Turner. We are grateful to Esra, Hudson, Paolo Ferraris and Jonathan Campbell for their comments on an earlier draft. 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.[4] In *Proc. NMR-2000*, 2000.

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

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

[Erdem *et al.*, 2000] Esra Erdem, Vladimir Lifschitz, and Martin Wong. Wire routing and satisfiability planning. In *Proc. CL-2000*, pages 822–836, 2000.

[Geffner, 2000] Hector Geffner. Functional Strips: a more flexible language for planning and problem solving. In Jack Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer, 2000. To appear.

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

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

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

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

[Kautz and Walser, 1999] Henry Kautz and Joachim Walser. State-space planning by integer optimization. In *Proc. AAAI-99*, pages 526–533, 1999.

[Koehler, 1998] Jana Koehler. Planning under resource constraints. In *Proc. ECAI-98*, pages 489–493, 1998.

[Lifschitz *et al.*, 2000] Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: The oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer, 2000. To appear.

[Lifschitz, 1997] Vladimir Lifschitz. Two components of an action language. *Annals of Mathematics and Artificial Intelligence*, 21:305–320, 1997.

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

[Lifschitz, 2000] Vladimir Lifschitz. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.

[Marek and Truszczyński, 1999] 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.

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

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

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

[McCarthy, 1999] John McCarthy. Elaboration tolerance.[5] In progress, 1999.

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

[Simons, 1999] Patrik Simons. Extending the stable model semantics with more expressive rules. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 305–316, 1999.

[Wolfman and Weld, 1999] Steven Wolfman and Daniel Weld. The LPSAT engine and its application to resource planning. In *Proc. IJCAI-99*, pages 310–316, 1999.

[Zhang, 1997] Hantao Zhang. An efficient propositional prover. In *Proc. CADE-97*, 1997.

Appendix

```
% File 'additive'

:- include 'arithmetic'.
```

[5] `http://www-formal.stanford.edu/jmc/elaboration.html` .

```
:- sorts
  fluent(integer)
    >> additiveFluent
    >> (additiveIFluent; additiveDZFluent);
  fluent(nnInteger)
    >> nnAdditiveFluent
    >> (nnAdditiveIFluent; nnAdditiveDZFluent).


:- variables
  V_A,V_A1,V_A2,V_A3,V_A4            :: action;
  V_I,V_I1                           :: integer;
  V_NNI                              :: nnInteger;
  V_AF                               :: additiveFluent;
  V_AF1                              :: additiveIFluent;
  V_AF2                              :: additiveDZFluent;
  V_NNAF                             :: nnAdditiveFluent;
  V_NNAF1                            :: nnAdditiveIFluent;
  V_NNAF2                            :: nnAdditiveDZFluent;
  V_X                                :: computed.

% fluent 'contribution'

:- constants
  contribution(action,additiveFluent)   :: fluent(integer);
  contribution(action,nnAdditiveFluent) :: fluent(integer).

default contribution(V_A,V_AF) eq 0.
default contribution(V_A,V_NNAF) eq 0.

% parsing propositions involving 'increments' and 'decrements'

:- op(1010,xfx,increments).
:- op(1010,xfx,decrements).
:- op(1015,xfx,by).

% definitions of 'increments' and 'decrements'
% in terms of 'causes'

:- macros
  #1 increments #2 by #3
    -> #1 causes contribution(#1,#2) eq #3;
  #1 increments #2 by #3 if #4
    -> #1 causes contribution(#1,#2) eq #3 if #4;
  #1 decrements #2 by #3
    -> #1 causes contribution(#1,#2) eq V_X
              if V_X is -(#3);
  #1 decrements #2 by #3 if #4
    -> #1 causes contribution(#1,#2) eq V_X
              if #4 && V_X is -(#3).

% total order on actions induced by @<

:- macros
  next(#1,#2) -> (#1 @< #2 &&
                   -(\/V_A2: (#1 @< V_A2 && V_A2 @< #2)));
  first(#1) -> -(\/V_A3: (V_A3 @< #1));
  last(#1) ->  -(\/V_A4: (#1 @< V_A4)).

% recursive definition of the sum of contributions
% over an initial segment of the set of actions

:- constants
  accumulatedContribution(action,additiveFluent)
                                     :: fluent(integer);
  accumulatedContribution(action,nnAdditiveFluent)
                                     :: fluent(integer).

caused accumulatedContribution(V_A,V_AF) eq V_I
   if first(V_A) && contribution(V_A,V_AF) eq V_I.

caused accumulatedContribution(V_A,V_AF) eq V_X
   if next(V_A1,V_A)
   && accumulatedContribution(V_A1,V_AF) eq V_I1
   && contribution(V_A,V_AF) eq V_I && sum(V_X,V_I,V_I1).

caused accumulatedContribution(V_A,V_NNAF) eq V_I
    if first(V_A) && contribution(V_A,V_NNAF) eq V_I.

caused accumulatedContribution(V_A,V_NNAF) eq V_X
   if next(V_A1,V_A)
   && accumulatedContribution(V_A1,V_NNAF) eq V_I1
   && contribution(V_A,V_NNAF) eq V_I && sum(V_X,V_I,V_I1).
```

```
% computing values of additive fluents

caused V_AF1 eq V_X
   if last(V_A) && accumulatedContribution(V_A,V_AF1) eq V_I
   after V_AF1 eq V_I1 && sum(V_X,V_I,V_I1).

caused V_AF2 eq V_I
   if last(V_A) && accumulatedContribution(V_A,V_AF2) eq V_I.

caused V_NNAF1 eq V_X
   if last(V_A) && accumulatedContribution(V_A,V_NNAF1) eq V_I
   after V_NNAF1 eq V_NNI && sum(V_X,V_I,V_NNI) && V_X>=0.

caused V_NNAF2 eq V_NNI
   if last(V_A) && accumulatedContribution(V_A,V_NNAF2) eq V_NNI.
```