A Representation of the Zoo World in the Language of the Causal Calculator

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

Hudson Turner
Computer Science Department
University of Minnesota at Duluth
hudson@d.umn.edu

Abstract

The Zoo World is an action domain proposed by Erik Sandewall as part of his Logic Modelling Workshop. We show how to represent this domain in the input language of the Causal Calculator.

1 Introduction

The Logic Modelling Workshop[1] is an environment for communicating axiomatizations of action domains of nontrivial size, created by Erik Sandewall. At the time of this writing, it contains informal descriptions of two domains—the Zoo World and the Traffic World. In this paper we show how to represent the first of these domains in the input language of the Causal Calculator (CCALC)[2]—an implementation of the nonmonotonic causal logic from [McCain and Turner, 1997]. A formalization of the Traffic World in the same language is presented in [Akman *et al.*, 2001]. The part of the language of CCALC used in both papers is closely related to action language $\mathcal{C}+$ introduced in [Giunchiglia *et al.*, 2001].

The next section contains an extensive quote from the Logic Modelling Workshop description of the Zoo World. Then we describe the syntax of the action language $\mathcal{C}+$ (Section 3) and relate it to CCALC (Section 4), present our formalization (Section 5), discuss how closely it corresponds to the informal description of the Zoo (Section 6) and show how we tested it (Section 7).

2 Sandewall's Description of the Zoo World

Here is the Logic Modelling Workshop description that we want to formalize:

The ZOO is a scenario world containing the main ingredients of a classical zoo: cages, animals in the cages, gates between two cages as well as gates between a cage and the exterior. In the ZOO world there are animals of several species, including humans. Actions in the world may include movement within and between cages, opening and closing gates, feeding the animals, one animal killing and eating another, riding animals, etc.

...A finite surface area consists of a large number of *positions*. For example, one may let each position be a square, so that the entire area is like a checkerboard. However, the exact shape of the positions is not supposed to be characterized, and the number of neighbors of each position is left open, except that each position must have at least one neighbor. The neighbor relation is symmetric, of course, and the transitive closure of the neighbor relation reaches all positions.

One designated location is called the *outside*; all other locations are called *cages*... The distinction between a 'large' number of positions and a 'small' number of locations suggests in particular that locations can be individually named under a unique names assumption, that every location is thus named, but on the other hand that at most a few of the positions are named, and that the number of positions is left unspecified in every scenario.

Each position is included in exactly one location. Informally, each cage as well as the outside consists of a set of locations[3], viewed for example as tiles on the floor. Two locations are neighbors if there is one position in each that are neighbors.

The scenario also contains a small number (in the same sense as above) of gates. Informally, these are to be thought of as gates that can be opened and closed, and that allow passage between a cage and the outside, or between two gates[4]. Formally, each gate is associated with exactly two positions that are said to be at its *sides*, and these positions must belong to different locations.

...Some designated animals will need to be named, but the set of animals in a scenario may be large, and it may not be possible to know them all or to name them all. Animals may be born and may die off over time.

Each animal belongs to exactly one of a number of species. All the species are named and explicitly known. The membership of an animal in a species does not change over time. The species *human* is always defined, and there is at least one human-species animal in each scenario.

Each animal also has the boolean properties *large* and *adult*. Some species are large, some are not. Adult members of large species are large animals; all other animals are small (non-large).

Each animal has a position at each point in time. Two

[1] `http://www.ida.liu.se/ext/etai/lmw/` .

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

[3] Apparently "positions" is meant here.

[4] Apparently "cages" is meant.

large animals can not occupy the same position, except if one of them rides on the other (see below).

...Animals can move. In one unit of time, an animal can move to one of the positions adjacent to its present one, or stay in the position where it is. Moves to non-adjacent positions are never possible. Movement is only possible to positions within the same location (for example, within the same cage), and between those two positions that are to the side of the same gate, but only provided the gate is open. Several animals can move at the same time.

Movement actions must also not violate the occupancy restriction: at most one large animal in each position. This restriction also holds within the duration of moves, in the sense that a concurrent move where animal A moves into a position at the same time as animal B moves out of it, is only possible if at least one of A and B is a small animal.

This means in particular that two large animals can not pass through a gate at the same time (neither in the same direction nor opposite directions).

...The following actions can be performed by animals...

- Move To Position. Can be performed by any animal, under the restrictions described above, plus the restriction that a human riding an animal can not perform the Move-To-Position action (compare below).
- Open Gate. Can be performed by a human when it is located in a position to the side of the gate, and has the effect that the gate is then open until the next time a Close Gate action is performed.
- Close Gate. Can be performed by a human when it is located in a position to the side of the gate, and has the effect that the gate is closed until the next time an Open Gate action is performed.
- Mount Animal. Can be performed by a human mounting a large animal, when the human is in a position adjacent to the position of the animal. The action fails if the animal moves at the same time, and in this case the result of the action is that the human moves to the position where the animal was. If successful, the action results in a state where the human rides the animal. This condition holds until the human performs a Getoff action or the animal performs a Throwoff action.

  When a human rides an animal, the human can not perform the Move action, and its position is the same as the animal's position while the animal moves.
- Getoff Animal to Position. Can be performed by a human riding an animal, to a position adjacent to the animal's present position provided that the animal does not move at the same time. Fails if the animal moves, and in this case the rider stays on the animal.
- Throwoff. Can be performed by an animal ridden by a human, and results in the human no longer riding the animal and ending in a position adjacent to the animal's present position. The action is nondeterministic since the rider may end up in any such position. If the resultant position is occupied by another large animal then the human will result in riding that animal instead.

3 Action Language $\mathcal{C}+$

Action languages are formal models of the parts of natural language that are used to talk about actions and their effects [Gelfond and Lifschitz, 1998]. We will describe the syntax of the action language $\mathcal{C}+$ introduced in [Giunchiglia *et al.*, 2001]—an extension of the language $\mathcal{C}$ from [Giunchiglia and Lifschitz, 1998] that includes symbols for non-Boolean fluents.

A *multi-valued propositional signature* is a set of symbols called *constants*, along with a nonempty set $Dom(c)$ of symbols assigned to each constant c. We call $Dom(c)$ the *domain* of c. If the domain of c is $\{\mathbf{f},\mathbf{t}\}$ then we say that c is *Boolean*. An *atom* of a signature σ is an expression of the form $c{=}v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. If a constant c is Boolean, it can be used as shorthand for the atom $c{=}\mathbf{t}$. A *formula* of σ is a propositional combination of atoms.

An *interpretation* of σ is a function that maps every element of σ to an element of its domain. An interpretation I *satisfies* an atom $c{=}v$ if $I(c){=}v$. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives.

Consider a multi-valued signature σ partitioned into *fluent symbols* σ^{fl} and *action symbols* σ^{act}. Intuitively, states of the world are interpretations of σ^{fl}, and actions are interpretations of σ^{act}. In what follows, by "formula" we will mean a formula of σ. A *state formula* is a formula of σ^{fl}. In many cases (including the formalization of the Zoo World below), all action symbols are Boolean; then each of them represents, intuitively, an "atomic" action, and executing an action a is understood as executing concurrently all atomic actions c for which $a(c) = \mathbf{t}$.

There are two kinds of propositions in $\mathcal{C}+$: *static laws* of the form

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

and *dynamic laws* of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H\,,$$

where F and G are state formulas and H is a formula.

Abbreviations have been introduced for many special kinds of propositions [Gelfond and Lifschitz, 1998], [Giunchiglia *et al.*, 2001]. For instance, for any Boolean action symbol c, state formula F and formula H,

$$c \textbf{ causes } F \textbf{ if } H$$

is shorthand for the dynamic law

$$\textbf{caused } F \textbf{ if } \top \textbf{ after } c{=}\mathbf{t} \wedge H\,.$$

A dynamic law of the form

$$c \textbf{ causes } \bot \textbf{ if } H$$

can be further abbreviated as

$$\textbf{nonexecutable } c \textbf{ if } H\,.$$

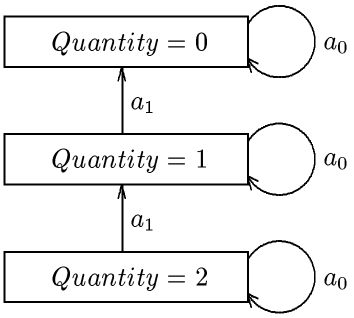

Figure 1: The transition diagram represented by propositions (1) for $N = 2$.

For any state formula F,

$$\textbf{inertial } F$$

stands for the dynamic law

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

It expresses that F "tends to remain true." Such propositions can be used to solve the frame problem.

The semantics of $\mathcal{C}+$ tells us how a set of propositions determines a "transition system"—a directed graph whose vertices correspond to states, and whose edges correspond to the execution of actions (see [Giunchiglia *et al.*, 2001, Section 4.3] for details).

As an example, let *Quantity* be a fluent symbol with the domain $\{0, \ldots, N\}$, where N is a fixed positive integer, and let *SubtractOne* be a Boolean action symbol. Consider the transition system described by the propositions

$$\begin{array}{ll} \textbf{inertial } \mathit{Quantity} = i & (i = 0, \ldots, N) \\ \mathit{SubtractOne} \textbf{ causes } \mathit{Quantity} = i - 1 & \\ \qquad \textbf{if } \mathit{Quantity} = i & (i = 1, \ldots, N) \\ \textbf{nonexecutable } \mathit{SubtractOne} \textbf{ if } \mathit{Quantity} = 0 & \end{array} \tag{1}$$

It has $N + 1$ states, corresponding to the possible values of *Quantity* (Figure 1). Since there is a single, Boolean action symbol *SubtractOne*, there are two actions: a_0 such that $a_0(\mathit{SubtractOne}) = \mathbf{f}$, and a_1 such that $a_1(\mathit{SubtractOne}) = \mathbf{t}$. To execute a_0 means, intuitively, to do nothing; this action is executable in every state. Action a_1 is executable when $\mathit{Quantity} \neq 0$.

4 Input Language of the Causal Calculator

The input language of CCALC allows us to express, among other things, propositions of $\mathcal{C}+$, along with declarations explaining the roles of the symbols used in these propositions—which of them are fluent symbols and which are action symbols, and what their domains are. When a set of propositions is specified using metavariables, such as i in (1), the language allows us to specify the ranges of these metavariables. Both domains of constants and ranges of metavariables are described in terms of "sorts."

File `subtract.t` (Figure 2) is the counterpart of (1) in the input language of CCALC. It illustrates some

```
% File 'subtract.t'

:- sorts
  number >> positiveNumber.

:- constants
  0             :: number;
  1..n          :: positiveNumber;
  quantity      :: inertialFluent(number);
  subtractOne   :: action.

:- variables
  I             :: positiveNumber;
  J             :: number.

subtractOne causes quantity eq J
            if quantity eq I && J is I-1.

nonexecutable subtractOne if quantity eq 0.
```

Figure 2: Example (1) expressed in the language of CCALC.

of the syntactic features used in the CCALC representation of the Zoo World in the next section. By writing `number >> positiveNumber` we express that `positiveNumber` is a subsort of `number`. The expression `inertialFluent(number)` in the declaration of `quantity` tells us that `quantity` is a fluent symbol whose domain is `number`, and that the propositions in the first line of (1) are postulated along with the propositions shown in the file explicitly. The symbol `&&` denotes conjunction. In the process of grounding, the expression `J is I-1` is replaced by logical constant `true` if the result of evaluating `I-1` equals the value of `J`, and by `false` otherwise.

As a computational tool, CCALC can be used (among other things) to answer queries about the effects of actions described in $\mathcal{C}+$ and to solve planning problems involving such actions. For instance, the following input file instructs CCALC to change the value of *Quantity* from 10 to 8 by performing the action *SubtractOne* not more than 3 times. (Like other satisfiability planners, CCALC requires that an upper bound on the number of steps be specified as part of a planning problem.)

```
:- macros
  n -> 10.

:- include 'subtract.t'.

:- plan
facts ::
  0: quantity eq 10;
goals ::
  3: quantity eq 8.
```

The macro expansion mechanism of CCALC is used here to specify the value of the parameter `n`. The symbols `0:` and `3:` are "time stamps": `quantity eq 10` is assumed to hold at time 0, and `quantity eq 8` is the goal to be achieved at time 3.

```
%%% ZOO LANDSCAPE %%%

:- sorts
  position;
  location >> cage;
  gate.

:- variables
  P,P1,P2                           :: position;
  L,L1,L2                           :: location;
  C                                 :: cage;
  G,G1                              :: gate.

:- constants
 % Each position is included in exactly one location (lmw)
  loc(position)                     :: fluent(location);

  neighbor(position,position)       :: defaultFalseFluent;
  reachable(position,position)      :: defaultFalseFluent;
  side1(gate)                       :: fluent(position);
  side2(gate)                       :: fluent(position);
  opened(gate)                      :: inertialFluent.
```

(Some of these fluents do not actually depend on the state of the world, as postulated in the next five propositions. In a fluent declaration, the sort is not specified when the fluent is Boolean.)

Table 1: Formalization

The solution found by CCALC (Version 1.9) is to perform the action *SubtractOne* between times 0 and 1, then again between times 1 and 2, and to do nothing between times 2 and 3. The output shows these actions, along with the value of *Quantity* at each step:[5]

```
0:  quantity eq 10

ACTIONS:  subtractOne

1:  quantity eq 9

ACTIONS:  subtractOne

2:  quantity eq 8

3:  quantity eq 8
```

CCALC can be used also to solve other computational problems related to actions. For instance, we can ask CCALC whether the set of assumptions about fluents and actions included in a given scenario is consistent. Like planning, answering queries of this kind is performed by invoking a propositional satisfiability solver.

5 Formalization

Our formalization of the Zoo World is shown in Tables 1–7 and is available online[6].

[5] Less trivial examples of planning problems solved by CCALC are discussed in [McCain and Turner, 1998], [Erdem *et al.*, 2000], [Lifschitz, 2000], [Lifschitz *et al.*, 2000] and [Lee and Lifschitz, 2001].

[6] `http://www.cs.utexas.edu/users/tag/cc/zoo.html` .

We distinguish between the general assumptions about the Zoo World quoted in Section 2 above, and specific details, such as the "topography" of the zoo (including the number of cages and gates), names of species other than `human`, and so forth. We formalize here the general assumptions only, and leave these details unspecified, just as the value of `n` is not specified in file `subtract.t` (Section 4). A description of all the specifics has to be added to our formalization to get an input file accepted by CCALC. The specific topography used in most of our computational experiments (Section 7) included 2 locations—a cage and the outside—that are separated by a gate and consist of 4 positions each:

```
--------
| 1  2 |  5  6
| 3  4 G  7  8
--------
```

The annotation `(lmw)` found in many comments below refers to the Logic Modelling Workshop description of the Zoo World quoted in Section 2.

6 Discussion

Some of the requirements quoted in Section 2 refer to the difference between "large numbers" (such as the number of positions) and "small numbers" (such as the number of locations), and between objects that can be "individually named" or "known" and objects that cannot. It seems to be impossible to address these distinctions in propositional languages such as $\mathcal{C}+$, and we have not tried. The only variables used in our formalization are metavariables ("schematic" variables); they range over syntactic objects, which are all "individually named."

```
constant loc(P) eq L.
constant neighbor(P,P1).
constant reachable(P,P1).
constant side1(G) eq P.
constant side2(G) eq P.
```

(In $\mathcal{C}+$, **constant** F is shorthand for a pair of dynamic laws expressing that the truth value of F never changes.)

```
% Each position must have at least one neighbor (lmw)
always (\/P1: neighbor(P,P1)).
```

(In $\mathcal{C}+$, **always** F is shorthand for **caused** $\perp$ **if** $\neg F$. In the language of CCALC, `\/` is the multiple disjunction symbol—the propositional counterpart of the existential quantifier.)

```
% The neighbor relation is irreflexive
never neighbor(P,P).
```

(In $\mathcal{C}+$, **never** F is shorthand for **caused** $\perp$ **if** F.)

```
% The neighbor relation is symmetric (lmw)
always neighbor(P,P1) ->> neighbor(P1,P).
```

(In the language of CCALC, `->>` is material implication.)

```
% Reachability is the reflexive transitive closure of the neighbor relation
caused reachable(P,P1) if neighbor(P,P1).
caused reachable(P,P).
caused (reachable(P,P1) && reachable(P1,P2)) ->> reachable(P,P2).
```

(These propositions have the intended meaning in view of the fact that `reachable` is declared to be false by default.)

```
% The transitive closure of the neighbor relation reaches all positions (lmw)
always reachable(P,P1).

:- constants
% One designated location is called the outside (lmw)
  outside           :: location.

% All other locations are cages (lmw)
always -(L=outside) ->> (\/C : (C=L)).
```

(In the language of CCALC, `-` is negation.)

```
:- macros
% Positions #1 and #2 are the two sides of gate #3
  sides(#1,#2,#3) -> ((side1(G) eq (#1) && side2(G) eq (#2))
                      ++ (side1(G) eq (#2) && side2(G) eq (#1)));
```

(In the language of CCALC, `++` is disjunction; `#1`, `#2`, etc., are understood as parameters by the macro expansion mechanism.)

```
% Two locations are neighbors if there is one position in each that are
%  neighbors (lmw)
  neighborl(#1,#2) -> (-((#1)=(#2))
       && (\/P1: \/P2: (loc(P1) eq (#1) && loc(P2) eq (#2) && neighbor(P1,P2)))).

% Each gate is associated with exactly two positions that are said to be at its
%  sides, and these positions must belong to different locations. (lmw)
never side1(G) eq P && side2(G) eq P1 && loc(P) eq L && loc(P1) eq L.

% No two gates have the same two sides
always (side1(G) eq P && side2(G) eq P1
        && side1(G1) eq P && side2(G1) eq P1) ->> G=G1.
```

Table 2: Formalization continued

```
% Two positions are neighbors if they are the sides of a gate
always sides(P1,P2,G) ->> neighbor(P1,P2).

% Two positions in different locations are neighbors only if they are the two
%  sides of a gate
always neighbor(P1,P2) && loc(P1) eq L1 && loc(P2) eq L2 && -(L1=L2)
                                           ->> (\/G: (sides(P1,P2,G))).

%%% ANIMALS %%%

:- sorts
  animal >> human;
  species.

:- variables
  ANML,ANML1                        :: animal;
  H,H1                              :: human;
  SP                                :: species.

:- constants
% Each animal belongs to exactly one of a number of species (lmw)
  sp(animal)                        :: fluent(species);
% One of the species is human (lmw)
  humanSpecies                      :: species;
% Some species are large, some are not (lmw)
  largeSpecies(species)             :: fluent;
% Each animal has a position at each point in time (lmw)
  pos(animal)                       :: inertialFluent(position);
% Boolean properties of animals (lmw)
  adult(animal)                     :: defaultFalseFluent;
  alive(animal)                     :: inertialFluent;

  mounted(human,animal)             :: inertialFluent.

% Membership of an animal in a species does not change over time (lmw)
constant sp(ANML) eq SP.

constant largeSpecies(SP).

% Humans are a species called humanSpecies
caused sp(H) eq humanSpecies.
always sp(ANML) eq humanSpecies ->> (\/H: (ANML=H)).

:- macros
% Adult members of large species are large animals (lmw)
  large(#1) -> adult(#1) && (\/SP: (sp(#1) eq SP && largeSpecies(SP))).

% There is at least one human-species animal in each scenario (lmw)
always (\/H: true).

% Animals cannot revive after death, or become young again once old
caused -alive(ANML) after -alive(ANML).
caused adult(ANML) after adult(ANML).

% Two large animals can not occupy the same position, except if one of them
%  rides on the other (lmw)
always pos(ANML) eq P && pos(ANML1) eq P && large(ANML) && large(ANML1)
       ->> (ANML=ANML1) ++ (\/H: (((H=ANML) && mounted(H,ANML1)) ++
                     ((H=ANML1) && mounted(H,ANML)))).
```

Table 3: Formalization continued

```
%%% CHANGING POSITION %%%

:- macros
  accessible(#1,#2) -> neighbor(#1,#2) && -(\/G: (sides(#1,#2,G) && -opened(G))).

% In one unit of time, an animal can move to one of the positions accessible
%  from its present one, or stay in the position where it is. Moves to non-
%  accessible positions are never possible (lmw)
caused false if pos(ANML) eq P1
   after pos(ANML) eq P && -((P=P1) ++ accessible(P,P1)).

:- constants
% Under "normal" circumstances, a large animal cannot occupy a position that,
%  at the previous time, was occupied by some other large animal, BUT there are
%  exceptions to this law of movement...
  abnormalEncroachment(human)       :: defaultFalseFluent.
```

(For the discussion of "abnormal encroachment" see Section 6.)

```
% A concurrent move where animal A moves into a position at the same time as
%  animal B moves out of it, is only possible if at least one of A and B is a
%  small animal. (lmw)   Exceptions for (failed) mount actions and certain
%  occurrences of throwOff -- when thrown human ends up where another large
%  animal was
caused false
   if pos(ANML) eq P && -(pos(ANML1) eq P)
      && (/\H: ((H=ANML) ->> -abnormalEncroachment(H)))
   after -(pos(ANML) eq P) && pos(ANML1) eq P
      && large(ANML) && large(ANML1).

% Two large animals cannot pass through a gate at the same time (neither in the
%  same direction nor opposite directions) (lmw)
caused false if pos(ANML) eq P1 && pos(ANML1) eq P1
   after pos(ANML) eq P && pos(ANML1) eq P && sides(P,P1,G)
         && large(ANML) && large(ANML1) && -(ANML=ANML1).
caused false if pos(ANML) eq P && pos(ANML1) eq P1
   after pos(ANML) eq P1 && pos(ANML1) eq P && sides(P,P1,G)
         && large(ANML) && large(ANML1) && -(ANML=ANML1).

% While a gate is closing, an animal cannot pass through it
caused false if pos(ANML) eq P1 && -opened(G)
   after pos(ANML) eq P && sides(P,P1,G) && opened(G).

%%% ACTIONS %%%

:- variables
  A,A1                              :: action.

:- constants
  move(animal,position), open(human,gate),
  close(human,gate), mount(human,animal),
  getOff(human,animal,position),
  throwOff(animal,human)            :: action.
```

Table 4: Formalization continued

```
:- macros
% Action #1 is executed by animal #2
  doneBy(#1,#2) -> ((\/P: (#1=(move(#2,P)))) ++
                  (\/G: ((#1=(open(#2,G))) ++ (#1=(close(#2,G))))) ++
                  (\/ANML: (#1=(mount(#2,ANML)))) ++
                  (\/ANML: \/P: (#1=(getOff(#2,ANML,P)))) ++
                  (\/H: (#1=(throwOff(#2,H))))).

% Every animal can execute only one action at a time
nonexecutable A && A1 if A@<A1 && doneBy(A,ANML1) && doneBy(A1,ANML1).
```

(In $\mathcal{C}+$, the conjunction of Boolean action symbols is usually interpreted as the concurrent execution of actions. In the language of CCALC, `@<` is a fixed total order.)

```
% Only live animals can execute actions
nonexecutable A if doneBy(A,ANML1) && -alive(ANML1).

% Direct effect of move action
move(ANML,P) causes pos(ANML) eq P.

% An animal can't move to the position where it is now
nonexecutable move(ANML,P) if pos(ANML) eq P.

% A human riding an animal cannot perform the move action (lmw)
nonexecutable move(H,P) if mounted(H,ANML).

% Effect of opening a gate
open(H,G) causes opened(G).

% A human cannot open a gate if it is not located at a position to the side of
%  the gate (lmw)
nonexecutable open(H,G)
            if pos(H) eq P && -(side1(G) eq P ++ side2(G) eq P).

% A human cannot open a gate if it is mounted on an animal
nonexecutable open(H,G) if mounted(H,ANML).

% A human cannot open a gate if it is already opened
nonexecutable open(H,G) if opened(G).

% Effect of closing a gate
close(H,G) causes -opened(G).

% A human cannot close a gate if it is not located at a position to the side of
%  the gate (lmw)
nonexecutable close(H,G)
             if pos(H) eq P && -(side1(G) eq P ++ side2(G) eq P).

% A human cannot close a gate if it is mounted on an animal
nonexecutable close(H,G) if mounted(H,ANML).

% A human cannot close a gate if it is already closed
nonexecutable close(H,G) if -opened(G).

% When a human rides an animal, its position is the same as the animal's
%  position while the animal moves (lmw)
caused pos(H) eq P if mounted(H,ANML) && pos(ANML) eq P.
```

Table 5: Formalization continued

```
% If a human tries to mount an animal that doesn't change position, mounting is
%  successful
caused mounted(H,ANML) if pos(ANML) eq P
                       after o(mount(H,ANML)) && pos(ANML) eq P.
```

(The syntax of the language of CCALC requires that, in some positions, action symbols be placed in the range of the operator `o`, which stands for "occurs.")

```
% The action fails if the animal changes position, and in this case the result
%  of the action is that the human ends up in the position where the animal was (lmw)
caused pos(H) eq P if -(pos(ANML) eq P)
                   after pos(ANML) eq P && o(mount(H,ANML)).

% A failed mount is not subject to the usual, rather strict, movement restriction
%  on large animals
mount(H,ANML) causes abnormalEncroachment(H).

% A human cannot attempt to mount a small animal (lmw)
nonexecutable mount(H,ANML) if -large(ANML).

% A human can only be mounted on a large animal
always mounted(H,ANML) ->> large(ANML).

% A human cannot attempt to mount a human who is mounted
nonexecutable mount(H,H1) if mounted(H1,ANML).

% A human cannot attempt to mount an animal on which another human is already mounted
nonexecutable mount(H,ANML) if mounted(H1,H).

% A human cannot be mounted on a human who is mounted
never mounted(H,H1) && mounted(H1,ANML).

% A human already mounted on some animal cannot attempt to mount another
nonexecutable mount(H,ANML) if mounted(H,ANML1).

% A human cannot attempt to mount an animal already mounted by a human
nonexecutable mount(H,ANML) if mounted(H1,ANML).

% An animal can be mounted by at most one human at a time
never mounted(H,ANML) && mounted(H1,ANML) && -(H=H1).

% A large human cannot attempt to mount another human
nonexecutable mount(H,H1) if large(H).

% A large human cannot be mounted on another human
never mounted(H,H1) && large(H).

% The getOff action is successful provided that the animal does not move at the
%  same time. It fails if the animal moves, and in this case the rider stays on
%  the animal (lmw)
caused pos(H) eq P if pos(ANML) eq P1
                   after o(getOff(H,ANML,P)) && pos(ANML) eq P1.

caused -mounted(H,ANML) if pos(ANML) eq P1
                   after o(getOff(H,ANML,P)) && pos(ANML) eq P1.
```

Table 6: Formalization continued

```
% The action cannot be performed by a human not riding an animal (lmw)
nonexecutable getOff(H,ANML,P) if -mounted(H,ANML).

% A human cannot attempt to getOff to the position that is not accessible from
%  the current position
nonexecutable getOff(H,ANML,P1) if pos(ANML) eq P && -accessible(P,P1).

% The throwOff action results in the human no longer riding the animal and ending
%  in a position adjacent to the animal's present position. It is nondeterministic
%  since the rider may end up in any position adjacent to the animal's present
%  position (lmw)
throwOff(ANML,H) may cause pos(H) eq P.
throwOff(ANML,H) causes -mounted(H,ANML).

% If the resultant position is occupied by another large animal then the human
%  will result in riding that animal instead (lmw)
caused mounted(H,ANML1)
   if pos(ANML1) eq P && large(ANML1) && pos(H) eq P && -(H=ANML1)
   after o(throwOff(ANML,H)).

% If the position a large human is thrown into was previously occupied by another
%  large animal, the usual movement restriction doesn't apply
throwOff(ANML,H) causes abnormalEncroachment(H).

% The action cannot be performed by an animal not ridden by a human (lmw)
nonexecutable throwOff(ANML,H) if -mounted(H,ANML).

% The actions getOff and throwOff cannot be executed concurrently
nonexecutable getOff(H,ANML,P) && throwOff(ANML,H).
```

Table 7: The end of the formalization

The first paragraph quoted in Section 2 also mentions actions—"feeding the animals" and "one animal killing and eating another"—not further described in the specification. We did not include those actions in our axiomatization, nor did we introduce any actions not mentioned in the specification. The specification also mentions, in passing, the possibility of "vehicles" moving about the zoo; we did not consider this.

With those exceptions, we tried to follow the specification closely. Nonetheless, we imposed a number of additional commonsense stipulations, such as the following.

- The neighbor relation is irreflexive.
- Positions in different locations are neighbors only if they are sides of the same gate.
- No two gates have the same sides.
- Only live animals can perform actions.
- An animal can't move to a position it's already in.
- A human can't open (close) a gate that is already opened (closed).

We also added a number of action preconditions that might conceivably be relaxed, among them the following.

- Each animal can execute at most one action at a time.
- A mounted human can't open or close a gate.
- A human cannot (attempt to) mount a human that is in turn already mounted on an animal.

In fact, we added many such preconditions for the mount action. (After all, it's a zoo, not a circus.) Such action preconditions are accompanied by corresponding state constraints when needed, prohibiting, for instance, initial states in which several humans are mounted on a single animal.

We also allow only small humans to mount humans. This restriction may seem reasonable enough in itself, but we in particular wanted to rule out the possibility that two humans could attempt to mount one another, simultaneously, with the result that they would wind up switching positions. This illustrates a fairly general kind of difficulty. Another example is this: We do not allow an animal to throw off a rider at the same time that the rider is getting off. Intuitively, it might be the case that the animal intends to throw the rider at precisely the moment that the rider intends to dismount, but, according to our formalization, only one of those two actions can in fact occur—either the animal manages to throw the rider, or the rider manages to dismount. There is an even simpler example of this kind: intuitively, two or more agents may intend to move to a given position at a given time. Our formalization reflects the idea that, despite their intentions, at most one of them can succeed! In such cases, we don't attempt to describe more complicated scenarios in which the agents may, for instance, "partially" succeed in executing the actions they intend to execute.

The Zoo specification explicitly leaves it to the axiomatizer to decide the following: What happens when a rider is thrown to a position that another large animal is either entering or leaving? We have chosen to have the thrown rider end up riding the animal only if it is entering the position. Otherwise, the thrown rider ends up (unmounted) in a position just vacated by a large animal. This latter possibility violates one of the general restrictions on movement, which we discuss next.

There are several general laws of movement in the specification of the Zoo World.

- At most one large animal in each position, unless one is mounted on the other.
- In a single time step, an animal's movements cannot take it further than the positions neighboring its current position.
- Animals cannot pass through closed gates.
- Two large animals cannot pass through a gate at the same time.
- A large animal cannot enter a position just vacated by a large animal.

We stipulate in addition that no animal can pass through a closing gate. It may be interesting to note that these restrictions are expressed, in our formalization, without mention of actions. Instead, we write dynamic laws that mention only fluents. The many action preconditions that they imply can then be left implicit.

There are exceptions to the general law that a large animal cannot enter a position just vacated by a large animal. The first exception is clearly required by the specification—a failed mount results in the human occupying the position vacated by the animal that was unsuccessfully mounted. The other exception is the one mentioned previously, involving a rider who is thrown into a position just vacated by a large animal. To accommodate these exceptions to the general law, we introduce a special-purpose abnormality fluent, `abnormalEncroachment`, which is false by default, but is made true whenever these exceptional circumstances arise.[7] The proposition expressing the general law is then formulated to take these exceptions into account via the abnormality fluent. That is, a large animal can enter a position just vacated by a large animal only when `abnormalEncroachment` holds.

Finally, the Zoo formalization listed here effectively defines `reachable` as the reflexive transitive closure of the `neighbor` relation. This allows us to express the requirement that all positions be reachable from all others. But this definition is costly to work with, and so, while correctly computed by CCALC, it is impractical. Under the current implementation, all propositions mentioning `reachable` should be dropped before using the system to answer queries.

In practice, the ability to answer queries with an automated system was, of course, quite useful during the formalization process, making it easier to refine and check our description.

[7] Test 3 of Section 7 shows the cases when this fluent becomes true.

7 Examples

To test our formalization, we asked CCALC whether certain scenarios are possible, and checked that its answers were in agreement with what we had expected. Here are some examples.

1. Can two animals, one large and one small, be in the same position? — Yes, this would not be in conflict with the occupancy restriction. In this test, we postulated that Homer is an adult human, Snoopy is a dog, humans are a large species, and dogs are a small species. CCALC determined that the following is consistent:

```
0: \/P: (pos(homer) eq P && pos(snoopy) eq P).
```

2. Can two large animals be in the same position? — The occupancy restriction disallows this, with one exception: one animal can be a human mounted on the second animal. CCALC found that the following is impossible:

```
0: \/P: (pos(homer) eq P && pos(jumbo) eq P
         && -mounted(homer,jumbo)).
```

Jumbo is an adult elephant (Figure 3).

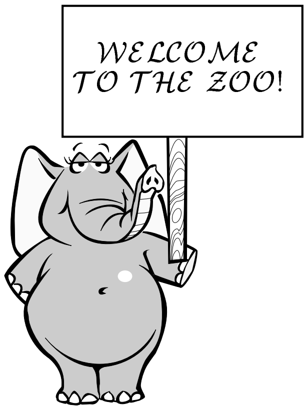

Figure 3: Jumbo

3. Can a large animal move into a position at the same time as another large animal moves out of it? — Yes. Although the occupancy restriction applies within the duration of moves, this scenario is possible in the process of a failed attempt of the first animal to mount the second. There is also the possibility that the first animal is thrown off into the position just vacated by the second.

To investigate this, we asked CCALC whether the following is possible:

```
\/P: ((0: -(pos(homer) eq P)) &&
      (1: pos(homer) eq P) &&
      (0: pos(jumbo) eq P) &&
      (1: -(pos(jumbo) eq P))),
0: mounted(homer,silver).
```

(Silver is a horse.) CCALC determined that it is, and found a solution in which Silver throws off Homer. Then we asked

```
\/P: ((0: -(pos(homer) eq P)) &&
      (1: pos(homer) eq P) &&
      (0: pos(jumbo) eq P) &&
```

```
    (1: -(pos(jumbo) eq P))),
 0: /\ANML: -o(throwOff(ANML,homer)).
```

CCALC found a solution in which Homer tries to mount Jumbo.

4. Can a large animal move into a position at the same time as another large animal moves out of it, given that neither animal is a human? — No, the occupancy restriction disallows this. CCALC answered no to the query:

```
 \/P: ((0: -(pos(silver) eq P)) &&
       (1: pos(silver) eq P) &&
       (0: pos(jumbo) eq P) &&
       (1: -(pos(jumbo) eq P))).
```

5. Can two large animals pass through a gate at the same time? — No. CCALC determined, for instance, that Homer and Jumbo cannot simultaneously pass through a gate in different directions:

```
 0: opened(gateAO),
 0: \/P: (pos(homer) eq P && loc(P) eq outside),
 0: \/P: (pos(jumbo) eq P && loc(P) eq cageA),
 1: \/P: (pos(homer) eq P && loc(P) eq cageA),
 1: \/P: (pos(jumbo) eq P && loc(P) eq outside).
```

6. Is it possible for a human to mount a large animal that is in a position adjacent to his current position? — Yes.

```
 0: /\P: /\P1:
      (pos(homer) eq P && pos(jumbo) eq P1
                          ->> neighbor(P,P1)),
 0: o(mount(homer,jumbo)),
 1: mounted(homer,jumbo).
```

7. Is it possible for a human to mount a large animal that is in a position not adjacent to his current position? — No.

```
 0: /\P: /\P1:
      (pos(homer) eq P && pos(jumbo) eq P1
                          ->> -neighbor(P,P1)),
 0: o(mount(homer,jumbo)),
 1: mounted(homer,jumbo).
```

8. Initially Homer is in a cage, with the gate closed. His goal is to get out; after that, the gate should be closed again. Can this happen by time 2? By time 3? — The earliest this can happen is time 3, because Homer needs to open the gate, pass through it, and close it again. This happens in the plan that CCALC generated in response to

```
facts ::
 0: -opened(gateAO),
 0: \/P: (pos(homer) eq P && loc(P) eq cageA);
goals ::
 2..3: (-opened(gateAO) &&
    \/P: (pos(homer) eq P && loc(P) eq outside)).
```

The pair of time stamps `2..3` instructs CCALC to look for a plan of length 2, and, if there is no such plan, to try length 3.

Acknowledgements

A logic program related to the Zoo World was written by several members of the Texas Action Group in September of 1999,[8] and discussing that program with Michael Gelfond helped us in our work on this paper in many ways. We are grateful to Varol Akman, Selim Erdoğan and anonymous referees for useful comments. The second author was partially supported by NSF under grant IIS-9732744. The third author was partially supported by NSF under CAREER Grant 0091773.

[8] `http://www.cs.utexas.edu/users/vl/tag/zoo_discussion` .

References

[Akman *et al.*, 2001] Varol Akman, Selim Erdoğan, Joohyung Lee, and Vladimir Lifschitz. A Representation of the Traffic World in the Language of the Causal Calculator.[9] To appear in *Common Sense 2001*.

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

[Gelfond and Lifschitz, 1998] Michael Gelfond and Vladimir Lifschitz. Action languages.[10] *Electronic Transactions on AI*, 3:195–210, 1998.

[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.

[Giunchiglia *et al.*, 2001] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Causal laws and multi-valued fluents.[11] Unpublished draft, 2001.

[Lee and Lifschitz, 2001] Joohyung Lee and Vladimir Lifschitz. Additive fluents.[12] In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[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*, pages 147–165. Kluwer, 2000.

[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.

[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.

[9] `http://www.cs.utexas.edu/users/vl/mypapers/traffic.ps` .

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

[11] `http://www.cs.utexas.edu/users/vl/mypapers/clmvf-long.ps` .

[12] `http://www.cs.utexas.edu/users/vl/mypapers/additive.ps` .