Representing the Zoo World and the Traffic World in the Language of the Causal Calculator

Varol Akman

Bilkent University, Ankara, Turkey

Selim T. Erdoğan, Joohyung Lee, Vladimir Lifschitz

University of Texas, Austin, USA

Hudson Turner

University of Minnesota, Duluth, USA

Abstract

The work described in this report is motivated by the desire to test the expressive possibilities of action language $\mathcal{C}+$. The Causal Calculator (CCALC) is a system that answers queries about action domains described in a fragment of that language. The Zoo World and the Traffic World have been proposed by Erik Sandewall in his Logic Modelling Workshop—an environment for communicating axiomatizations of action domains of nontrivial size.

The Zoo World consists of several cages and the exterior, gates between them, and animals of several species, including humans. Actions in this domain include moving within and between cages, opening and closing gates, and mounting and riding animals. The Traffic World includes vehicles moving continuously between road crossings subject to a number of restrictions, such as speed limits and keeping a fixed safety distance away from other vehicles on the road. We show how to represent the two domains in the input language of CCALC, and how to use CCALC to test these representations.

1 Introduction

The work described in this report is motivated by the desire to test the expressive possibilities of action language $\mathcal{C}+$ introduced in the companion paper [Giunchiglia *et al.*, 2003]. Specifically, we are interested in the expressiveness of the “definite” fragment of $\mathcal{C}+$ that is implemented in the Causal Calculator (CCalc).

Language $\mathcal{C}+$ is a recent addition to the family of formal languages for describing actions, which started with STRIPS [Fikes and Nilsson, 1971] and ADL [Pednault, 1994]. It is an extension of action language $\mathcal{C}$ defined in [Giunchiglia and Lifschitz, 1998]. The Causal Calculator uses propositional satisfiability solvers to answer queries about action domains described in $\mathcal{C}+$. Its original version was written by Norman McCain as part of his dissertation [McCain, 1997], and now it is being maintained by Texas Action Group at Austin.[1] It can be downloaded from its home page

`http://www.cs.utexas.edu/users/tag/ccalc/` .

CCalc has been applied to several challenge problems in the theory of commonsense knowledge [Lifschitz, 2000], [Lifschitz *et al.*, 2000], [Campbell and Lifschitz, 2003] and to the formalization of norm-governed computational systems [Artikis *et al.*, 2003].

The two test domains discussed in this paper, the Zoo World and the Traffic World, have been proposed by Erik Sandewall in his Logic Modelling Workshop (LMW)[2]—an environment for communicating axiomatizations of action domains of nontrivial size. The Zoo World consists of several cages and the exterior, gates between them, and animals of several species, including humans. Actions in this domain include moving within and between cages, opening and closing gates, and mounting and riding animals. The Traffic World includes vehicles moving continuously between road crossings subject to a number of restrictions, such as speed limits and keeping a fixed safety distance away from other vehicles on the road. More details can be found in the next section, which contains extensive quotes from the LMW descriptions of both domains.

In accordance with our goal, we have attempted to translate these descriptions into the input language of CCalc as closely as we could, including the elements that look somewhat arbitrary. One such element in the LMW description of the Zoo World has to do with the “occupancy restriction”—there can be at most one large animal in each position. On the one hand, LMW specifies that this restriction holds even dynamically: a concurrent move, where one animal moves into a position at the same time as another animal moves out of it, is only possible if at least one of the animals is small. On the other hand, the specification tells us that an attempt to mount an animal fails if the animal moves at the same time, in which case “the result of the action is that the human moves to the position where the animal was.” Thus a failed mount is an exception to the occupancy restriction. In view of this fact, the

[1] `http://www.cs.utexas.edu/users/tag` .
[2] `http://www.ida.liu.se/ext/etai/lmw/` .

occupancy restriction has to be formalized as a “defeasible” dynamic law. It is interesting to note that expressing such laws in $\mathcal{C}+$ calls for the use of “nonexogenous action constants”—a new feature of this language, not available in its ancestor $\mathcal{C}$.

After a brief introduction to the input language of CCalc in Section 3, we present and discuss a formalization of the Zoo World in Section 4 and a formalization of the Traffic World in Section 5. These two main parts of the paper do not logically depend on each other and can be read in any order.

2 The Logic Modelling Workshop Descriptions

2.1 The Description of the Zoo World

The following is the LMW description of the Zoo World 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 positions, 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 cages. 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 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 his 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.

2.2 The Description of the Traffic World

The following is the LMW description of the Traffic World that we want to formalize:

> The TRAFFIC scenario world is intended to capture simple hybrid phenomena: vehicles moving continuously with well defined velocities along roads with well defined lengths, respecting speed limits and other restrictions on the vehicle's behaviors.
>
> The landscape in the TRAFFIC Scenario World uses the following two types:
>
> - *Nodes*, which can be thought of as road crossings without any particular structure (no lanes, etc)
> - *Segments*, which can be thought of as road segments each of which connects two nodes.
>
> The set of nodes and the set of segments are both considered as fully known, and all nodes and segments can be assigned individual names.
>
> Each segment has exactly one *start node* and exactly one *end node*. It also has a *length*, which is a real number (or rational number, if preferred). This is all the structure there is.
>
> ...The activity structure in the TRAFFIC world uses only one sort:
>
> - *Cars*, which are intuitively thought of as driving along the arcs in the TRAFFIC landscape structure.
>
> Each car has a *position* at each point in time. The position is indicated as a pair consisting of the segment where the car is located, and the distance travelled along the segment. The distance travelled is a number between 0

and the segment's length.

Each car has a *top speed*, and each road segment has a *speed limit*. The actual velocity of a car at each point in time is the maximum velocity allowed by the following three conditions:

- The speed limit of the road segment where it is driving
- Its own top speed
- Surrounding traffic restrictions

Cars drive at piecewise constant velocity, and can change velocity discontinuously. (A more refined variant, TRAFFIC2, will require cars to change their velocity continuously, and assumes piecewise constant acceleration/deceleration). When a car arrives at a node ("intersection") then it may continue on any segment that connects to that node, except the one it is arriving at.

Cars can drive in both "directions" along a segment, that is, they can move both from the start node to the end node, and vice versa.

Cars can not overtake — if two cars go in the same direction on the same road segment, and one catches up with the other, then it has to stay behind at least until they arrive to the next node, where possibly the second car can choose another direction onwards. Cars going in opposite directions on the same segment can meet without difficulty, however.

The surrounding traffic restriction says that a car is never allowed to be closer than a fixed safety distance *varsigma* to the car in front of it, and it may never get itself into a situation where that could happen. This means, first of all, that when it gets to a distance of *varsigma* to a car moving in front of it on the same segment and in the same direction, then it must reduce speed to match the speed of the car in front of it. Also when getting close to a node (= an intersection), a car must reduce its speed in a way that takes into consideration all other cars that are just approaching or leaving the same node.

3 Language of the Causal Calculator

To understand the rest of this paper on the technical level, the reader will need some knowledge of the syntax and semantics of $\mathcal{C}+$. This information can be found, for instance, in Sections 1–2.3, 3.1–3.3 and 4.1–4.3 of [Giunchiglia *et al.*, 2003].

A CCalc input file consists of declarations, causal laws in the sense of $\mathcal{C}+$ (or, more often, schemas with metavariables whose instances are causal laws), queries (for instance, planning problems) and comments. A brief, informal introduction to the language of CCalc is given in [Giunchiglia *et al.*, 2003, Section 6]. Here are the features of CCalc which are not mentioned in that introduction but are used in our formalizations of the Zoo World and Traffic World.

1. The ASCII representations of some symbols used in the language of CCALC are summarized in the following chart:

Symbol	$\neg$	$\neq$	$\wedge$	$\vee$	$\supset$	$\equiv$	$\bot$	$\top$
ASCII representation	`-`	`\=`	`&`	`++`	`->>`	`<->`	`false`	`true`

Encoding multiple conjunctions and disjunctions by ASCII characters can be illustrated by this example:

$$\textbf{constraint} \bigvee_{P_1} neighbor(P, P_1)$$

is written in CCALC as

```
constraint [\/P1 | neighbor(P,P1)].
```

2. The symbol `>>` between the names of two sorts expresses that the second is a subsort of the first, so that every object that belongs to the second sort belongs also to the first. For instance, the declarations

```
:- sorts
  location >> cage.

:- objects
  cageA                  :: cage.
```

tell us that `cageA` is both a `cage` and a `location`.

3. If a sort name is composed by adding `+none` to the name of a previously declared sort, this means that the set of objects of that sort consists of the objects of the previously declared sort and the auxiliary symbol `none`. These "`+none`-sorts" are used for declaring partial valued fluent constants and are not explicitly declared as sorts.

For instance, assume that the objects of sort `node` are `a` and `b`:

```
:- objects
  a, b                   :: node.
```

Then the objects of sort `node+none` will be `a`, `b` and `none`. This sort is useful when we want to talk about the node at which a car currently is,

```
:- constants
  node(car)              :: sdFluent(node+none).
```

which may be none if the car is in the middle of a segment.

4. In the example above, note that the fluent constant node is declared as an sdFluent. This stands for "statically determined fluent constant" [Giunchiglia *et al.*, 2003, Sections 4.2, 5.5]. Their values are determined by static causal laws only and are not assumed exogenous in the initial state.

5. Rigid constants [Giunchiglia *et al.*, 2003, Section 4.6] can be declared as in the following example.

```
:- constants
  sp(animal)             :: species.
```

Function constant sp represents an operation that turns an animal into a species. The value of the constant is not affected by any event.

6. The arguments of constants are supposed to be objects; when a constant has another constant as an argument, the whole expression is understood as an abbreviation. For instance, the schema

```
  nonexecutable move(ANML,pos(ANML))
```

("an animal cannot move into its current position") has the same meaning as

```
  nonexecutable move(ANML,P) if pos(ANML)=P
```

where P is a position variable.

7. Macros can be declared as in the following example.

```
:- macros
  position(#1,#2,#3) -> (segment(#1)=(#2) & distance(#1)=(#3)).
```

(#1,#2,... are parameters for macros.) Upon reading an input file, CCALC replaces every occurrence of a pattern in the left-hand side of -> with the corresponding instance of the right-hand side.

8. In the process of grounding, some parts of a schema turn into 0-place connectives $\top$, $\bot$. For instance, grounding turns G=G1 in the schema

```
  constraint sides(P,P1,G) & sides(P,P1,G1) ->> G=G1.
```

into $\top$ when G and G1 are instantiated by the same object, and into $\bot$ otherwise.

9. A comment starts with a %.

4 The Zoo World

We present our formalization of the Zoo World along with detailed comments in Section 4.1, discuss how closely it corresponds to the informal description in Section 4.2, and show how we used CCalc to test it in Section 4.3.

4.1 Formalization

Our formalization of the Zoo World shown below is also available online[3].

We distinguish between the general assumptions about the Zoo World quoted in Section 2.1 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. 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 our computational experiments is described in Section 4.3.

The annotation `(lmw)` found in many comments below refers to the Logic Modelling Workshop description of the Zoo World quoted in Section 2.1.

```
%%% ZOO LANDSCAPE %%%

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

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

:- constants
% Each position is included in exactly one location (lmw)
  loc(position)                       :: location;
  neighbor(position,position)         :: boolean;
```

[3] `http://www.cs.utexas.edu/users/tag/ccalc/zoo/` .

```
  side1(gate)                          :: position;
  side2(gate)                          :: position;
  opened(gate)                         :: inertialFluent.

default -neighbor(P,P1).

% Each position must have at least one neighbor (lmw)
constraint [\/P1 | neighbor(P,P1)].

% The neighbor relation is irreflexive
constraint -neighbor(P,P).

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

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

% All other locations are cages (lmw)
constraint [\/C | L=C] where L\=outside.
```

(A `where` clause at the end of a schematic expression instructs CCalc to limit grounding to the values of the variables that satisfy the given test. This contributes to efficient grounding.)

```
% Two positions are the sides of a gate
:- constants
  sides(position,position,gate)        :: boolean.

caused sides(P,P1,G) if side1(G)=P & side2(G)=P1.
caused sides(P,P1,G) if side1(G)=P1 & side2(G)=P.
default -sides(P,P1,G).

% 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)
constraint loc(side1(G))\=loc(side2(G)).
```

(As in 6 of Section 3, the argument of `loc` is supposed to be an object, not a constant. Here `side1(G)`, `side2(G)` are understood to be the value of each constant.)

```
% No two gates have the same two sides
constraint sides(P,P1,G) & sides(P,P1,G1) ->> G=G1.
```

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

% Two positions in different locations are neighbors only if
%  they are the two sides of a gate
constraint loc(P)\=loc(P1) & neighbor(P,P1)
                  ->> [\/G | sides(P,P1,G)].


%%% ANIMALS %%%

:- sorts
  animal >> human;
  species.

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

:- objects
% One of the species is human (lmw)
  humanSpecies                      :: species.

:- constants
% Each animal belongs to exactly one of a number of species.
%  (lmw)  Membership of an animal in a species does not change
%  over time (lmw)
  sp(animal)                        :: species;
% Some species are large, some are not (lmw)
  largeSpecies(species)             :: boolean;
% Each animal has a position at each point in time (lmw)
  pos(animal)                       :: inertialFluent(position);
% Boolean properties of animals (lmw)
  adult(animal)                     :: boolean;
  mounted(human,animal)             :: inertialFluent.

default largeSpecies(SP).
default adult(ANML).

% Humans are a species called humanSpecies
caused sp(H)=humanSpecies.
constraint sp(ANML)=humanSpecies ->> [\/H | ANML=H].
```

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

% There is at least one human-species animal in each scenario
%  (lmw)
constraint [\/H | true].

% Two large animals can not occupy the same position,
%  except if one of them rides on the other (lmw)
constraint pos(ANML)=pos(ANML1) & large(ANML) & large(ANML1)
     ->> [\/H | (H=ANML & mounted(H,ANML1)) ++
                (H=ANML1 & mounted(H,ANML))] where ANML@<ANML1.
```

(`@<` is a fixed total order.)

```
%%% CHANGING POSITION %%%

:- constants
accessible(position,position)       :: sdFluent.

caused accessible(P,P1)
   if neighbor(P,P1) & -[\/G | sides(P,P1,G) & -opened(G)].
default -accessible(P,P1).

% In one unit of time, an animal can move to one of the posi-
%  tions accessible from its present one, or stay in the posi-
%  tion where it is.  Moves to non-accessible positions are
%  never possible (lmw)
constraint pos(ANML)\=P1
   after pos(ANML)=P & -(P=P1 ++ accessible(P,P1)).
```

(The proposition **constraint** F **after** G is defined in [Giunchiglia *et al.*, 2003, Appendix B] as an abbreviation for **caused** $\perp$ **if** $\neg F$ **after** G.)

```
% 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 (see the first two propositions in '%%% ACTIONS %%%')
constraint -(pos(ANML)=P & pos(ANML1)\=P)
   after pos(ANML)\=P & pos(ANML1)=P
         & large(ANML) & large(ANML1) unless ab(ANML).
```

(Appending the `unless` clause makes the causal law defeasible; under certain exceptional circumstances, this assertion is retracted. See [Giunchiglia *et al.*, 2003, Section 4.3].)

```
% Two large animals cannot pass through a gate at the same time
% (neither in the same direction nor opposite directions) (lmw)
constraint -(pos(ANML)=P1 & pos(ANML1)=P1)
   after pos(ANML)=P & pos(ANML1)=P & sides(P,P1,G)
         & large(ANML) & large(ANML1) where ANML@<ANML1.
constraint -(pos(ANML)=P & pos(ANML1)=P1)
   after pos(ANML)=P1 & pos(ANML1)=P & sides(P,P1,G)
         & large(ANML) & large(ANML1) where ANML@<ANML1.

% While a gate is closing, an animal cannot pass through it
constraint -opened(G) ->> pos(ANML)\=P1
   after pos(ANML)=P & sides(P,P1,G) & opened(G).


%%% ACTIONS %%%

:- variables
  A,A1                              :: exogenousAction.

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

:- 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)]).
```

(Different from "=" used in an atom, "==" is a comparison operator.)

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

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

(The two propositions above describe exceptional circumstances for the movement restriction between large animals.)

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

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

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

% 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 he is not located at a position
%  to the side of the gate (lmw)
nonexecutable open(H,G)
           if -(pos(H)=side1(G) ++ pos(H)=side2(G)).

% A human cannot open a gate if he 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 he is not located at a posi-
%  tion to the side of the gate (lmw)
nonexecutable close(H,G)
           if -(pos(H)=side1(G) ++ pos(H)=side2(G)).

% A human cannot close a gate if he 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, his position is the same as the
%  animal's position while the animal moves (lmw)
caused pos(H)=P if mounted(H,ANML) & pos(ANML)=P.

% If a human tries to mount an animal that doesn't change posi-
%  tion, mounting is successful
caused mounted(H,ANML) if pos(ANML)=P
                       after pos(ANML)=P & mount(H,ANML).

% 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)=P if pos(ANML)\=P
                after pos(ANML)=P & mount(H,ANML).

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

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

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

% A large human cannot be mounted on a human
constraint mounted(H,H1) ->> -large(H).

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

% An animal can be mounted by at most one human at a time
constraint -(mounted(H,ANML) & mounted(H1,ANML)) where H@<H1.

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

% A human cannot be mounted on a human who is mounted
constraint -(mounted(H,H1) & mounted(H1,ANML)).

% A human cannot attempt to mount an animal if the human is
```

```
%  already mounted by a human
nonexecutable mount(H,ANML) if mounted(H1,H).

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

% 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)=P if pos(ANML)=P1
                after pos(ANML)=P1 & getOff(H,ANML,P).

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

% 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 a position that is not
%  accessible from the current position
nonexecutable getOff(H,ANML,P) if -accessible(pos(ANML),P).

% 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 posi-
%  tion (lmw)
throwOff(ANML,H) may cause pos(H)=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(H)=pos(ANML1) & large(ANML1)
   after throwOff(ANML,H) where H\=ANML1.

% 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).
```

4.2 Discussion

Some of the requirements quoted in Section 2.1 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 the language of CCALC, which does not have quantifiers—they are simulated by multiple conjunctions and disjunctions. In a language like this, all objects are "individually named."

The specification quoted in Section 2.1 mentions the transitive closure of the neighbor relation (it is required to reach all positions). The causal rules mimicking the usual logic programming definition of the transitive closure do not provide a correct characterization of this concept [Giunchiglia *et al.*, 2003, Section 6.2], and we did not include a formalization of the requirement that refers to transitive closure. [4]

The specification mentions also several kinds of events that are not described in any detail—feeding animals, killing and eating animals, being born and dying. It also mentions, in passing, the possibility of "vehicles" moving about the zoo. We did not include any of these in our representation.

The specification defines two locations to be neighbors if there is one position in each that are neighbors. This extension of the neighbor relation to locations is not mentioned anywhere else, and we did not include it in our formalization. But it would be easy to represent this extension, either by a statically determined fluent constant or by a macro. For instance, we can write:

```
:- macros
  neighbor1(#1,#2) ->
      ((#1)\=(#2) & [\/P \/P1 | loc(P)=(#1) & loc(P1)=(#2)
                                            & neighbor(P,P1)]).
```

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.

[4] Transitive closure can be characterized by a causal theory that is not definite in the sense of [Giunchiglia *et al.*, 2003, Section 2.6], or by a definite causal theory using some auxiliary constants.

- 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 are 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 who 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, states in which several humans are mounted on a single animal.

We also allow only small humans (that is, children) 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 off 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 off, or the rider manages to dismount. There is an even simpler example of this kind: intuitively, two or more animals 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 that large animal only if it is entering the position. Otherwise, the thrown rider ends up (unmounted) in a position just vacated by the animal. This latter possibility is an exception to 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 fluent constants. 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 make the related proposition defeasible by appending the `unless` clause. The abnormality constant used in the clause is false by default, but is made true whenever these exceptional circumstances arise. [5] Under such circumstances, a large animal can enter a position just vacated by another large animal despite the movement restriction between large animals.

4.3 Testing

To test our formalization, we gave CCalc queries and checked that its answers matched our expectations. Queries related to action domains and their CCalc representations are discussed in [Giunchiglia *et al.*, 2003, Sections 3.3, 6]. Besides the representation of the Zoo World shown above, the CCalc input included the description of a specific landscape. The zoo we used for testing is small. It includes 2 locations—a cage and the outside—that are separated by a gate and consist of 4 positions each (Figure 1). All positions within the cage are each other's neighbors, as well as all outside positions. The input also included information about the specific animals mentioned in each query.

1. The gate is closed, and Homer, an adult human, is in position 6. *His goal is to mount Jumbo, an adult elephant, which is in position* 3 *and is not going to move around. How many steps are required to achieve this goal?*

This question can be represented by the following CCalc query:

[5] Test 4 (Section 4.3) shows the cases when this exception happens.

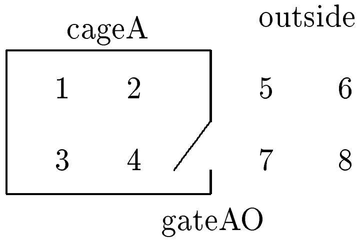

Fig. 1. A zoo landscape

```
:- query
  maxstep :: 3..4;
  0: -opened(gateAO),
      pos(homer)=6;
  maxstep: mounted(homer,jumbo);
  T=<maxstep ->> (T: pos(jumbo)=3).
```

(In the last line, `T` is a variable for the initial segment of integers—numbers from `0` to `10`.)

CCalc has determined that the length of the shortest solution is 4. It found a solution in which Homer walks to the gate, opens it, walks into the cage, and then mounts Jumbo.

2. The gate was closed, and Homer was outside; after two steps, he was inside. What can we say about his initial position?

To answer this question, we asked CCalc to find all models satisfying the conditions

```
  0: -opened(gateAO),
     loc(pos(homer))=outside;
  2: loc(pos(homer))=cageA.
```

CCalc has determined that Homer's only possible initial position is 7. Homer opened the gate and moved to position 4.

3. Initially Homer was outside, and Snoopy, a dog, was inside the cage, with the gate closed. Is it possible that they switched their locations in one step? in two steps? If the elephant Jumbo is substituted for Snoopy, will the answers be the same?

What is essential here is that small animals, unlike elephants, are not affected by the occupancy restriction (Section 2.1); Homer and Snoopy can pass through the gate simultaneously. In response to the query

```
:- query
  maxstep :: 1..2;
  0: -opened(gateAO),
     loc(pos(homer))=outside,
     loc(pos(snoopy))=cageA;
  maxstep:
     loc(pos(homer))=cageA,
     loc(pos(snoopy))=outside.
```

CCalc reported that the length of the shortest solution is 2. In case of Jumbo, CCalc surprised us by discovering that the length of the shortest solution is 4, and not 5 as we had thought. Homer opens the gate, mounts Jumbo (on the other side), dismounts (by either being thrown off or getting off), following which Jumbo moves out of the cage. When we told CCalc that Homer never mounts Jumbo, CCalc agreed that the length of the shortest possible sequence of actions is 5.

4. Can a large animal move into a position at the same time as another large animal moves out of it?

The answer is yes. Although the occupancy restriction applies within the duration of moves (Section 2.1), 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)=P)) &
         (1: pos(homer)=P) &
         (0: pos(jumbo)=P) &
         (1: -(pos(jumbo)=P))];
  0: mounted(homer,silver).
```

(Silver is a horse.) CCalc found a solution in which Silver throws off Homer. Then we replaced the last line of the query with

```
  0: [/\ANML| -throwOff(ANML,homer)].
```

CCalc found a solution in which Homer tried to mount Jumbo. On the other hand, a horse cannot possibly move into a position at the same time as an elephant moves out of it. Accordingly, CCalc determined that there is no model satisfying the condition

```
  [\/P | (0: -(pos(silver)=P)) &
         (1: pos(silver)=P) &
```

```
      (0: pos(jumbo)=P) &
      (1: -(pos(jumbo)=P))].
```

5. In position 1*, Jumbo throws off Homer. What are the possible positions of Jumbo and Homer after that?*

This question illustrates the nondeterministic character of the Throwoff action (Section 2.1). The given assumption can be represented by the condition

```
0: pos(jumbo)=1,
   throwOff(jumbo,homer).
```

According to CCALC, in the models satisfying this condition Homer is thrown into positions 2, 3 and 4; Jumbo always stays in position 1.

5 The Traffic World

In this section we present our formalization of the Traffic World. We start with discussing the possibility of describing continuous motion using integer arithmetic in Section 5.1 and the use of action languages for representing change in the absence of actions in Section 5.2. Our formalization of the Traffic World is presented in Section 5.3 and examples of the use of CCALC for testing the formalization are shown in Section 5.4. Finally, Section 5.5 compares our approach to the Traffic World with related work.

5.1 Continuous Motion and Integer Arithmetic

Under some special circumstances, questions about continuous motion can be discussed using integers, without ever mentioning reals or even rational numbers. Such special circumstances are assumed in our formalization of the Traffic World. Specifically, we assume that

- the lengths of all road segments and the safety distance *varsigma* are expressed by integers,
- the top speeds of all cars and the speed limits on all road segments are expressed by integers,
- in the scenarios under consideration, the times when cars leave and reach endpoints of road segments, and the times at which the distance between two cars traveling on the same segment in the same direction becomes *varsigma* are expressed by integers.

These constraints are similar to those adopted in the description of the spacecraft Integer in [Lee and Lifschitz, 2003]:

> 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 spacecraft Integer is actually close to the refined version of the Traffic World mentioned in Section 2.2 above, in the sense that its velocity changes continuously, and its acceleration is piecewise constant.

For query answering on the basis of our formalization of the Traffic World, it is essential that all time instants mentioned in the queries be expressed by integers.

Assumptions like these make it easier to describe motion in the action languages whose semantics is defined in the framework of transition systems. The use of these languages for describing motion without such simplifying assumptions is a topic for future research.

5.2 Change in the Absence of Actions

In our formalization of the Traffic World, the action of selecting a new road segment when a car reaches an intersection is denoted by $ChooseSegment(c, sg)$—the driver of car c chooses to turn into road segment sg.

This action of selecting a new road segment when a car reaches an intersection are actually the only action performed in the Traffic World. Between intersections every car is assumed to be moving at the maximum speed allowed by the road conditions, so that the drivers are not permitted to perform any actions affecting the speeds of their cars.

The speeds of cars may change many times, however, during a time interval that does not include the execution of actions. Consider, for instance, a long road segment with a high speed limit, and two cars moving along it in the same direction. If the top speed of the car in front is lower than the top speed of the second car then the latter will slow down at some point to match the speed of the slower car. If there are several cars on the road moving in the same direction then the surrounding traffic restriction may force the last of them to reduce its speed several times, although no actions will be executed.

(An example of such a scenario is shown in Section 5.4.)

A similar phenomenon will be observed when several cars have simultaneously (or almost simultaneously) approached the same node from different directions, with the intention to turn into the same road. The cars will have to leave the intersection one by one, after the intervals that will guarantee the safety distance between them. In our formalization, there are no causal laws determining the order in which the cars are going to depart. This nondeterministic sequence of events may take a long time, and it does not involve the execution of actions.

Change in the absence of actions, so essential in the Traffic World, can be described in $\mathcal{C}+$ using dynamic laws that do not contain action constants. For instance, the dynamic law

$$\begin{aligned}&\textbf{caused } \mathit{Distance}(c) = \mathit{ds} + \mathit{sp}\\ &\qquad\qquad \textbf{after } \neg \mathit{WillLeave}(c) \wedge \mathit{Distance}(c) = \mathit{ds} \wedge \mathit{Speed}(c) = \mathit{sp}\end{aligned}$$

says that if (i) car c is going to remain on the same road segment during the next time interval, (ii) the distance travelled by c along that segment so far is ds, and (iii) the speed of c during that time interval is going to be equal to sp, then by the end of the time interval the distance travelled by c along the current road segment will become $ds + sp$.

5.3 Formalization

Our formalization of the Traffic World is shown below and is available online [6].

We distinguish between the general assumptions about the Traffic World quoted in Section 2.2 above and specific details, such as the number and positions of road segments, the numerical values of their speed limits, the number of cars and their top speeds. Here we formalize only the general assumptions, and leave such details unspecified, as in the formalization of the Zoo World. A description of a specific scenario has to be added to our formalization to get an input file accepted by CCalc.

As with the Zoo World, the annotation `(lmw)` found in many comments below refers to the Logic Modelling Workshop description of the Traffic World quoted in Section 2.2.

[6] `http://www.cs.utexas.edu/users/tag/ccalc/traffic/` .

```
%%% TRAFFIC %%%

:- sorts
  integer;
  node;
  segment;
  car.

:- variables
  Nd                          :: node;
  Sg                          :: segment;
  C,C1                        :: car;
  Ds,Ds1                      :: integer;
  Sp,Sp1                      :: integer.

:- objects
  0..maxInt                   :: integer.
```

(Since the range of integers that we need depends on the scenario, we define an appropriate value for the macro `maxInt` in each scenario description.)

```
:- constants
% If a car is pointing toward the end node of the segment on
%  which it currently is, its positiveOrientation is true
  positiveOrientation(car)    ::  inertialFluent;

% Each car has a position at each point in time. The position is
%  indicated as a pair consisting of the segment where the car
%  is located, and the distance travelled along the segment
%  (lmw)
  segment(car)                :: inertialFluent(segment);
  distance(car)               :: simpleFluent(integer).

:- macros
  position(#1,#2,#3) -> (segment(#1)=(#2) & distance(#1)=(#3)).

:- constants
% Each segment has exactly one start node and exactly one end
%  node (lmw)
  startNode(segment)          :: node;
  endNode(segment)            :: node;

% Each segment has a length, which is a real number (or rational
%  number, if preferred). (lmw)  We assume it to be an integer
  length(segment)             :: integer;
```

```
% Each road segment has a speed limit (lmw)
  speedLimit(segment)        :: integer;

% Each car has a top speed (lmw)
  topSpeed(car)              :: integer;

% Actual speed of a car during the next time interval
  speed(car)                 :: sdFluent(integer);

% The new segment a car will continue on
  nextSegment(car)           :: inertialFluent(segment+none);

% A car will leave the segment on which it is currently
%  travelling
  willLeave(car)             :: simpleFluent;

% The node at which a car is
  node(car)                  :: sdFluent(node+none).

exogenous willLeave(C).

% The position of a car determines whether it is at a node or
%  not and, if it is, which node it is at.
caused node(C)=Nd
   if positiveOrientation(C) & position(C,Sg,0)
      & startNode(Sg)=Nd.
caused node(C)=Nd
   if -positiveOrientation(C) & position(C,Sg,0)
      & endNode(Sg)=Nd.
caused node(C)=Nd
   if positiveOrientation(C) & position(C,Sg,length(Sg))
      & endNode(Sg)=Nd.
caused node(C)=Nd
   if -positiveOrientation(C) & position(C,Sg,length(Sg))
      & startNode(Sg)=Nd.

default node(C)=none.

% A car will have a positive orientation after leaving the
%  start node of the segment it is entering
caused positiveOrientation(C)
   after willLeave(C) & nextSegment(C)=Sg
         & node(C)=startNode(Sg).
```

```
% A car will have a negative orientation after leaving the end
%  node of the segment it is entering
caused -positiveOrientation(C)
   after willLeave(C) & nextSegment(C)=Sg & node(C)=endNode(Sg).

% Proceed to the next segment if the car was about to leave
caused segment(C)=Sg after willLeave(C) & nextSegment(C)=Sg.

% The distance covered by a car which remained on the same
%  segment
caused distance(C)=Ds+Sp
   after -willLeave(C) & distance(C)=Ds & speed(C)=Sp
                                  where Ds+Sp=<maxInt.
```

(A `where` clause at the end of a schematic expression instructs CCalc to limit grounding to the values of the variables that satisfy the given test.)

```
% The distance covered by a car right after it changed to a new
%  segment
caused distance(C)=Sp after willLeave(C) & speed(C)=Sp.

% The time when a car reaches a node is assumed to be an integer
constraint position(C,Sg,Ds) ->> Ds=<length(Sg).

% No two cars on the same segment and having the same
%  orientation can be closer than varsigma (lmw)
constraint position(C,Sg,Ds) & position(C1,Sg,Ds1)
      ->> positiveOrientation(C)\=positiveOrientation(C1)
      where C@<C1 & abs(Ds1-Ds)<varsigma.
```

(`@<` is a fixed total order; `abs` stands for the absolute value.)

```
% If a car is waiting at a node and there is a car too close on
%  the next segment it will travel on, the time at which the car
%  in front will reach a distance of varsigma from the node is
%  assumed to be an integer
constraint position(C1,Sg,Ds1) ->> Ds1=<varsigma
   after position(C1,Sg,Ds) & Ds<varsigma & nextSegment(C)=Sg
         & modifiedOrientation(C)=positiveOrientation(C1).

:- constants
% For any car in the middle of a segment, its modifiedOrienta-
%  tion has the same value as its positiveOrientation.  If a car
%  has selected a new segment, then modifiedOrientation has the
%  value that positiveOrientation would have if the car were at
```

```
%  the beginning of the new segment
  modifiedOrientation(car)   :: sdFluent.

caused modifiedOrientation(C)
   if nextSegment(C)=none & positiveOrientation(C).
caused modifiedOrientation(C)
   if nextSegment(C)=Sg & node(C)=startNode(Sg).

default -modifiedOrientation(C).
```

(The proposition **constraint** F **after** G is defined in [Giunchiglia *et al.*, 2003, Appendix B] as an abbreviation for **caused** $\bot$ **if** $\neg F$ **after** G.)

```
:- constants
% The relation between modifiedSegment and segment is similar
  modifiedSegment(car)       :: sdFluent(segment).

caused modifiedSegment(C)=Sg if nextSegment(C)=none
                                    & segment(C)=Sg.
caused modifiedSegment(C)=Sg if nextSegment(C)=Sg.

:- constants
% The relation between modifiedDistance and distance is similar
  modifiedDistance(car)      :: sdFluent(integer).

caused modifiedDistance(C)=Ds if nextSegment(C)=none
                                    & distance(C)=Ds.
caused modifiedDistance(C)=0 if nextSegment(C)\=none.

:- constants
% Maximum speed allowed by the top speed of a car and the speed
%  limit of the segment on which the car will be travelling
  maxSpeed(car)             :: sdFluent(integer).

caused maxSpeed(C)=min(Sp,Sp1)
   if topSpeed(C)=Sp & nextSegment(C)=none
      & speedLimit(segment(C))=Sp1.
```

(As in 6 of Section 3, the argument of `speedLimit` is supposed to be an object, not a constant. Here `segment(C)` is understood to be equal to the value of that constant.)

```
caused maxSpeed(C)=min(Sp,Sp1)
   if topSpeed(C)=Sp & nextSegment(C)=Sg
      & speedLimit(Sg)=Sp1.
```

```
:- constants
% Maximum distance a car can cover on the segment on which it
%  will be travelling
  maxDistance(car)           :: sdFluent(integer).

caused maxDistance(C)=Ds+Sp
   if modifiedDistance(C)=Ds & maxSpeed(C)=Sp
                          where Ds+Sp=<maxInt.

:- constants
% The first car is ahead of the second
  ahead(car,car)             :: sdFluent.

caused ahead(C1,C)
   if positiveOrientation(C1)=modifiedOrientation(C)
      & segment(C1)=modifiedSegment(C)
      & distance(C1)>=modifiedDistance(C) where C\=C1.
default -ahead(C1,C).

% No overtaking
constraint -ahead(C,C1) after ahead(C1,C).

:- constants
% The first car is ahead of the second car and not farther
%  than varsigma from it
  varsigmaAhead(car,car)           :: sdFluent.

caused varsigmaAhead(C1,C)
   if ahead(C1,C) & distance(C1)=Ds1 & modifiedDistance(C)=Ds
                                       where Ds1-Ds=<varsigma.
default -varsigmaAhead(C1,C).

% The actual velocity of a car at each point in time is the max-
%  imum velocity allowed by the following three conditions:
%   - The speed limit of the road segment where it is driving
%   - Its own top speed
%   - Surrounding traffic restrictions (lmw)

% If a car is in the middle of a segment and there is no other
%  car which is varsigma ahead of the car and which will not
%  leave, then it will travel at its maximum speed
caused speed(C)=Sp
   if nextSegment(C)=none & maxSpeed(C)=Sp
      & [/\C1 | varsigmaAhead(C1,C) ->> willLeave(C1)].
```

```
% If a car is in the middle of a segment and there is a car
%  varsigma ahead of it which will not leave, then its speed
%  will be the smaller of its maximum speed and the speed of
%  the car in front
caused speed(C)=min(Sp,Sp1)
   if nextSegment(C)=none & maxSpeed(C)=Sp
      & varsigmaAhead(C1,C) & -willLeave(C1) & speed(C1)=Sp1.

% If a car is at the end of a segment and will not leave then
%  it will stay where it is
caused speed(C)=0 if nextSegment(C)\=none & -willLeave(C).

% If a car is at the end of a segment and will enter a new seg-
%  ment where there is no car within varsigma, then it will
%  travel at its maximum speed
caused speed(C)=Sp
   if willLeave(C) & maxSpeed(C)=Sp
      & [/\C1 | -varsigmaAhead(C1,C)].

% If a car is at the end of a segment and will enter a new seg-
%  ment where there is a car within varsigma, its speed will be
%  the smaller of its maximum speed and the speed of the car in
%  front
caused speed(C)=min(Sp,Sp1)
   if willLeave(C) & varsigmaAhead(C1,C)
      & maxSpeed(C)=Sp & speed(C1)=Sp1.

:- constants
% Choose a new segment for a car to proceed
  chooseSegment(car,segment) :: exogenousAction.

% Direct effect of choosing a new segment
chooseSegment(C,Sg) causes nextSegment(C)=Sg.

% Cannot choose the segment that is already chosen
nonexecutable chooseSegment(C,Sg) if nextSegment(C)=Sg.

% When a car arrives at a node ("intersection") then it may
%  continue on any segment that connects to that node... (lmw)
constraint node(C)=startNode(Sg) ++ node(C)=endNode(Sg)
   after chooseSegment(C,Sg).

% ...except the one it is arriving at (lmw)
constraint nextSegment(C)\=Sg after segment(C)=Sg.
```

```
% A car cannot arrive at a node without choosing a new segment
%  on which to proceed
constraint -[\/Nd | node(C)=Nd]
   after [/\Sg | -chooseSegment(C,Sg)] & node(C)=none.

% A car can't have a next segment unless it has travelled to the
%  end of its current segment
caused nextSegment(C)=none if node(C)=none.
caused nextSegment(C)=none
   if node(C)=startNode(segment(C)) & positiveOrientation(C).
caused nextSegment(C)=none
   if node(C)=endNode(segment(C)) & -positiveOrientation(C).

% Only cars which have selected a new segment can leave
constraint willLeave(C) ->> nextSegment(C)\=none.

% At most one car will leave a node and enter a new segment at
%  each time
constraint willLeave(C) & willLeave(C1) & node(C)=node(C1)
           ->> nextSegment(C)\=nextSegment(C1) where C@<C1.

% If there is a car at a node which has selected a new segment,
%  and there are no cars within varsigma from the node, then
%  there should be a car which will leave the node (i.e. no
%  unnecessary waiting is allowed)
constraint nextSegment(C)=Sg &
           -[\/C1 | varsigmaAhead(C1,C) & distance(C1)<varsigma]
    ->> [\/C1 | node(C1)=node(C) & nextSegment(C1)=Sg
                & willLeave(C1)].
```

5.4 Examples

We tested certain aspects of our formalization by giving CCalc queries about several scenarios and checking that its answers matched our expectations. Besides the representation of the Traffic World shown above, the input for each scenario included the description of a specific landscape. Here are three examples:

1. Consider three roads with the lengths and speed limits shown in Figure 2. The top speed of `car1` *is 2 and it is initially located at node* `a`*. (Note that segment* `seg_bc1` *is long and has a high speed limit whereas segment* `seg_bc2` *is short but its speed limit is low.) Which way must the car go in order to*

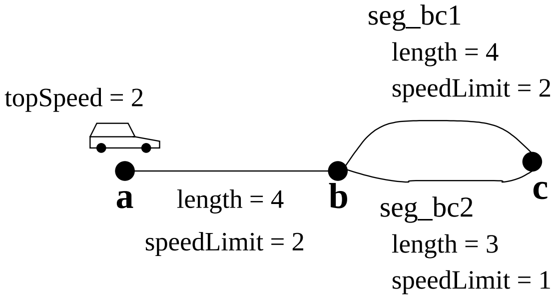

Fig. 2. Initial state and landscape for Example 1

reach `c` *from* `a` *as soon as possible?*

This planning problem can be described as follows:

```
:- query
  maxstep :: 1..10;
  0: position(car1,seg_ab,0),
     positiveOrientation(car1);
  maxstep: node(car1)=c.
```

CCALC found the following shortest plan:

```
0:  distance(car1)=0  segment(car1)=seg_ab  speed(car1)=2
    node(car1)=a

1:  distance(car1)=2  segment(car1)=seg_ab  speed(car1)=2

ACTIONS:  chooseSegment(car1,seg_bc1)

2:  willLeave(car1)  distance(car1)=4  segment(car1)=seg_ab
    nextSegment(car1)=seg_bc1  speed(car1)=2  node(car1)=b

3:  distance(car1)=2  segment(car1)=seg_bc1  speed(car1)=2

ACTIONS:  chooseSegment(car1,seg_bc2)

4:  willLeave(car1)  distance(car1)=4  segment(car1)=seg_bc1
    nextSegment(car1)=seg_bc2  speed(car1)=1  node(car1)=c
```

In order to be at node `c` at time 4, the car needs to take the segment with the higher speed limit. Another variant of this problem asked CCALC to find a plan in which `car1` would be at node `c` at time 5. In that case the plan returned showed `car1` choosing to take `seg_bc2`.

Also note in the plan above that just before the car arrived at node `c`, it chose a new segment. This is because of the proposition that prevents a car from arriving at a node without concurrently choosing a new segment.

2. The top speeds of three cars and their initial positions are as shown in the top diagram of Figure 3. Cars `car1`, `car2` *and* `car3` *have top speeds of 3, 2 and 1, respectively. Assuming the speed limit is 3 and the safety distance is 1, how will the cars move during the next 3 time intervals?*

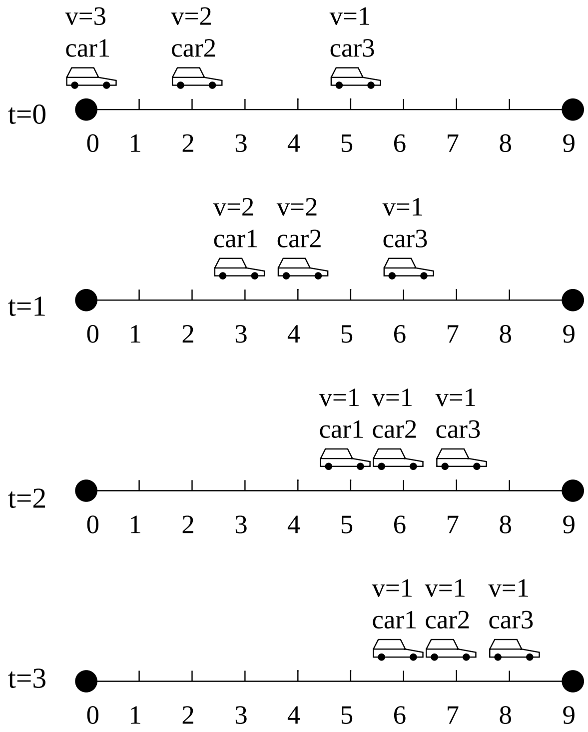

Fig. 3. Example 2

The conditions can be represented by the following query:

```
:- query
  maxstep  :: 3;
  0: position(car1,seg_ab,0),
     positiveOrientation(car1),
     position(car2,seg_ab,2),
     positiveOrientation(car2),
     position(car3,seg_ab,5),
     positiveOrientation(car3).
```

CCalc found a scenario satisfying the conditions (Figure 3):

```
0:  distance(car1)=0  distance(car2)=2  distance(car3)=5
    speed(car1)=3  speed(car2)=2  speed(car3)=1
```

```
1:  distance(car1)=3  distance(car2)=4  distance(car3)=6
    speed(car1)=2  speed(car2)=2  speed(car3)=1

2:  distance(car1)=5  distance(car2)=6  distance(car3)=7
    speed(car1)=1  speed(car2)=1  speed(car3)=1

3:  distance(car1)=6  distance(car2)=7  distance(car3)=8
    speed(car1)=1  speed(car2)=1  speed(car3)=1
```

3. The landscape and the initial positions of two cars are shown at the top of Figure 4. The top speeds of both cars and the speed limits for all segments are 1. The safety distance is 2. What are the possible scenarios in which both cars will be on segment `seg_cd` *at time 5?*

This question can be represented by the following query:

```
:- query
  maxstep :: 5;
  0: position(car1,seg_ac,1),
     position(car2,seg_bc,1),
     [/\C | positiveOrientation(C)];
  maxstep: [/\C | segment(C)=seg_cd].
```

CCalc has determined that there are two possibilities. Here is the first (Figure 4):

```
0:  distance(car1)=1  distance(car2)=1  segment(car1)=seg_ac
    segment(car2)=seg_bc  speed(car1)=1  speed(car2)=1

1:  distance(car1)=2  distance(car2)=2  segment(car1)=seg_ac
    segment(car2)=seg_bc  speed(car1)=1  speed(car2)=1

ACTIONS:  chooseSegment(car1,seg_cd)  chooseSegment(car2,seg_cd)

2:  willLeave(car1)  distance(car1)=3  distance(car2)=3
    segment(car1)=seg_ac  segment(car2)=seg_bc
    nextSegment(car1)=seg_cd  nextSegment(car2)=seg_cd
    speed(car1)=1  speed(car2)=0  node(car1)=c  node(car2)=c

3:  distance(car1)=1  distance(car2)=3  segment(car1)=seg_cd
    segment(car2)=seg_bc  nextSegment(car2)=seg_cd  speed(car1)=1
    speed(car2)=0  node(car2)=c

4:  willLeave(car2)  distance(car1)=2  distance(car2)=3
```

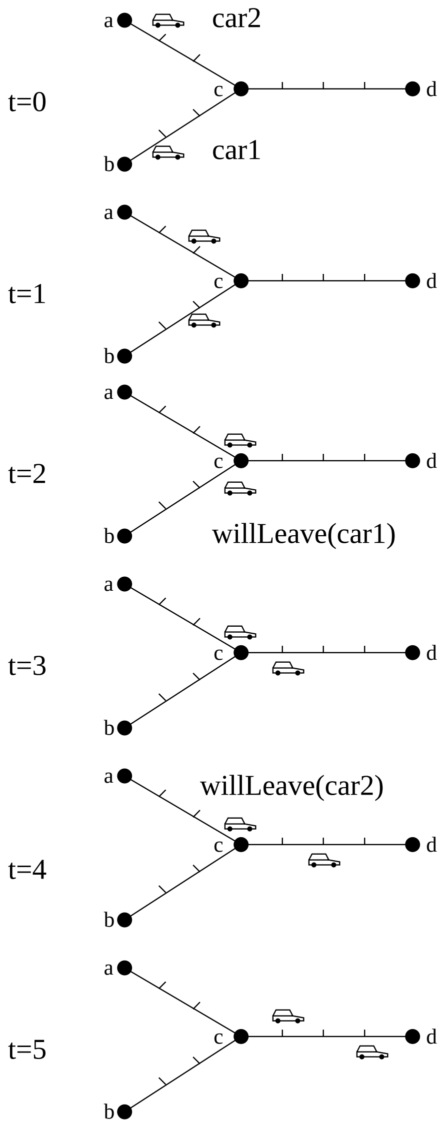

Fig. 4. Example 3, Scenario 1

```
    segment(car1)=seg_cd  segment(car2)=seg_bc
    nextSegment(car2)=seg_cd  speed(car1)=1  speed(car2)=1
    node(car2)=c

5:  distance(car1)=3  distance(car2)=1  segment(car1)=seg_cd
    segment(car2)=seg_cd  speed(car1)=1  speed(car2)=1
```

In the second scenario, the states at times 0 and 1 and the actions performed between times 1 and 2 are the same. For times 2 through 5, everything is the same except that `car1` is replaced by `car2` everywhere, and vice versa.

The fact that the same actions lead to different states illustrates the nondeterminism in the transition system to which the formalization corresponds. This nondeterminism arises when more than one car is waiting to enter a new segment. In such situations the car which will leave the node is chosen non-deterministically so, as in this example, there may be multiple scenarios that differ from each other by the order of cars leaving the node.

5.5 Related Work

Henschel and Thielscher [2000] showed how to formalize the Traffic World in the fluent calculus [Thielscher, 1998]. They do not assume that speeds and lengths are expressed by integers, as we do (Section 5.1). Other than that, their representation differs from ours in the following ways:

- Instead of having all cars obey the same rules, cars are divided into two groups: "deliberative" cars and "non-deliberative" cars. Drivers of deliberative cars can set their own speeds (using an action to change the speed of a car) instead of having to go at the maximum speed possible, and are allowed to wait at nodes. Non-deliberative cars are just like cars in our paper, as described in the Logic Modelling Workshop specification (Section 2.2).
- There is a "waiting area" associated with each node-segment pair. In this area, the cars that are going to enter the segment wait in line until there are no cars within *varsigma* from the node. When the new segment becomes free, the first car in the waiting area is allowed to leave.
- At any time, a car is in exactly one of three states: moving on a segment, at a node, or in a waiting area. Cars which are at a node or in a waiting area are not considered to be on segments. So after a car arrives at a node, the cars following it are allowed to arrive at the node too, even if the car remains there.
- For each node, the segments leading to it are assigned priorities. When several drivers simultaneously decide to turn into the same segment, their cars are placed in the waiting area in the order determined by the priorities of the segments they arrived from.

If a car is considered to be on some road segment at all times, as in our formalization, the number of cars that can approach a node is limited by the number of roads leading to that node. The view adopted in [Henschel and Thielscher, 2000], on the other hand, makes a car at a node exempt from the surrounding traffic restriction, so that the number of cars that can gather at a node (or in a waiting area) is unlimited. Perhaps we model streets in a city, and Henschel and Thielscher model roads between towns. It would not be difficult to modify our formalization to include deliberative cars and waiting areas; see [Erdoğan, 2000, Section 5.3] for a related discussion.

6 Conclusion

This paper described how two interesting and quite different medium-sized domains, the Zoo World and the Traffic World, can be represented in the input language of the Causal Calculator. The Zoo World contains many different actions, agents (animals and humans), and interactions between those agents but does not involve much continuous action. The Traffic World, by contrast, has only one action, but it involves continuous change in the states of many agents of the same kind (cars in this case). It also involves change in the absence of actions.

Two interesting aspects of action domains, nondeterminism and indirect effects of actions, are parts of both Zoo and Traffic and are handled conveniently in their representations in the input language of CCalc. (Throwing off a mounted human and the order in which cars leave segments are examples of nondeterminism in Zoo and Traffic, respectively. A rider being in the position of the animal which he mounts and changing of the speeds of cars after choosing a new segment are examples of indirect effects in the two domains.) In fact, indirect effects are quite easy to handle thanks to the ability to write static laws, which represent how the values of certain fluents affect the values of other fluents without reference to any actions or time.

The formalization of the Zoo World shows how exceptions to generally valid laws can be represented using "defeasible" dynamic laws. Some laws can be rendered ineffective under certain conditions.

In the formalization of the Traffic World, continuous motion is described using integers, under certain simplifying assumptions. Change without any actions can be easily described using dynamic laws that do not involve actions.

Despite the convenience with which most parts of the specifications could be represented, there are two aspects of the domains for which the input language of CCalc was not completely suitable. The specification of the Zoo World mentions the distinction between small numbers of objects which can all be named and 'large' numbers of object which cannot all be named. Due to the absence of real quantifiers in CCalc, such large sets of objects cannot be represented. The closest approximation is to use multiple disjunctions or conjunctions. Another shortcoming of CCalc is that it can only deal with integers. Therefore certain assumptions had to be made about the nature of the continuous motion in the Traffic World, restricting the representation.

As a final note, this work was motivated by the desire to test the expressive possibilities of $\mathcal{C}+$, which is a formal language with a precisely defined semantics. But having a working practical system, such as CCalc, was a great advantage when trying to formalize a complicated domain. It is quite easy to write flawed causal laws, and having an automated system to compare our intuitions with the actual effects of the causal laws was helpful. We were often corrected by CCalc and sometimes surprised when it answered our queries with results that we had not expected.

Acknowledgements

A logic program related to the Zoo World was written by several members of Texas Action Group in September of 1999,[7] and discussing that program with Michael Gelfond helped us in our work on this paper in many ways. Varol Akman was partially supported by TÜBİTAK-NSF under Grant 101E024. Selim Erdoğan, Joohyung Lee and Vladimir Lifschitz were partially supported by the National Science Foundation under Grant IIS-9732744 and by the Texas Higher Education Coordinating Board under Grant 003658-0322-2001. Hudson Turner was partially supported by the National Science Foundation under CAREER Grant 0091773.

References

[Artikis *et al.*, 2003] A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *Proceedings of Workshop on Agent-Oriented Software Engineering III (AOSE)*, LNCS 2585. Springer, 2003.

[Campbell and Lifschitz, 2003] Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning.[8] In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

[Erdoğan, 2000] Selim T. Erdoğan. Formalization of the TRAFFIC world in the $\mathcal{C}$ action language.[9] Master's thesis, Bilkent University, 2000.

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

[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.

[7] `http://www.cs.utexas.edu/users/vl/tag/zoo_discussion` .
[8] `http://www.cs.utexas.edu/users/vl/papers/sams.ps` .
[9] `http://www.cs.utexas.edu/users/selim/publications/tezdeneme.ps.gz`
.

[Giunchiglia *et al.*, 2003] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories.[10] *Artificial Intelligence*, 2003. To appear.

[Henschel and Thielscher, 2000] Andreas Henschel and Michael Thielscher. The LMW Traffic world in the fluent calculus.[11] *Linköping Electronic Articles in Computer and Information Science ISSN 1401-9841*, 5(014), 2000.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Describing additive fluents in action language $\mathcal{C}+$[12] In *Proc. IJCAI-03*, 2003. To appear.

[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, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[13] PhD thesis, University of Texas at Austin, 1997.

[Pednault, 1994] Edwin Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4:467–512, 1994.

[Thielscher, 1998] Michael Thielscher. Introduction to the fluent calculus.[14] *Electronic Transactions on AI*, 3, 1998.

[10] `http://www.cs.utexas.edu/users/vl/papers/nmct.ps` .

[11] `http://www.ep.liu.se/ea/cis/2000/014/` .

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

[13] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z` .

[14] `http://www.ep.liu.se/ea/cis/1998/014/` .