An experiment using DLV-K

Rocío Santillán and Alejandra López

23rd March 2004

Abstract

The Traffic World is an action domain proposed by Erik Sandewall. We represented part of this domain in the logic-based planning language DLV-K. Throughout this article we discuss the pros and cons of our approach using this language.

1 Introduction

In this paper we present the results about our experiment in DLV K. This software is an implementation of non monotonic logic reasoning, a logic-based planning language. More information about the background of this software can be found in [1]. Vladimir Lifschitz and his group modeled the problem in another planning language called CCALC. Their paper [2] was useful for our model of the problem. When we model the problem we faced some problems because of the structure that DLV-K specified. These problems will be explained in detail later.

2 Problem posing

The original problem defines attemps to abstract common behavior of vehicular traffic. The vehicles are moving along roads, where a road is provided with a defined length and a speed limit. The scenario that is defined by Sandewall has the following entities: Roads are represented as a graph. Each node is a crossroad and edges are segments of roads. Edges have start and end nodes. An edge has a length and speed limit. The only vehicles represented in the model are cars. They have a constant speed limit and they are associated with a position and velocity at each point in time.

The restrictions established between the objects are:

1. When a car is arriving to a node, it should reduce its speed and at the node its speed must be zero. It must take into consideration all other cars in the same node before continuing his way.
2. A car must reduce its velocity when it is behind another car, its actual speed is larger than the other's and they are very close. This is because in the original definition cars can not overtake.

3. When a car arrives to a node, it stops and chooses a new segment which must be different than the one it just left.
4. Cars can move from the start to the end node but also in the other direction of the segment. Two cars can be in the same position and time but in opposite directions.
5. Car's velocity is calculated at each point in time as the minimum between its top speed, the speed limit of the road and considering the previous traffic restrictions.
6. When cars are in a node, at most one car will leave the node at each time.

The entities defined in our model are:

- Cars have a top speed, a position in some segment, an actual speed and an orientation.
- Nodes.
- Segments have a start and end node, a speed limit and a length. Each entity has also an identifier. All the numerical attributes are defined as nonnegative integers.

We only present some of the rules that are in the original definition.

- When a car goes from the start to the end node, it has a positive orientation, in the other case its orientation is negative. (Restriction 4)
- To define actual speed, it's considered only the car's top speed, the segment's speed limit and if the end of segment is close. (Restrictions 1,5)
- Cars select a segment when they arrive at a node. The segment must be different than the one they come from. (Restriction 3)

The input of the problem will be the initial state of each car and the final state that we hope each car will arrive. The output expected is a representation of states in the time that model the movement of cars as real as posible.

3 Modeling the problem in DLV-K

Background knowledge is the part of the program where we can define some necesary aspects for the input of the problem. We kept it in a different file.

Background knowledge models the static part of the problem like entities. Also, it includes functions which are independent of the problem and are necessary to obtain information to find the solution.

First we define the attributes of the car, its name and top speed:

```
car(c1, 3).
car(c2, 5).
```

After that, we define nodes or crossroads and we assign them a name or identifier.

```
node(a).
node(b).
node(c).
node(d).
```

The segments have an identifier, a start node, a final node, a length and a speed limit.

```
segment(a_b,a,b,7,4).
segment(c_b,c,b,5,2).
segment(b_d,b,d,10,3).
```

Also we define a function to get the minimum between two values. This function was very useful to get information for the kind of solution we implemented.

```
min(V1,V2,V1):-V1<V2, #int(V1), #int(V2).
min(V1,V2,V2):-V2<=V1, #int(V1), #int(V2).
```

The second part of the software defines the possible dynamic states, which are the knowledge that could be modified throughout the program. In this case we define the position of the car in a segment, its orientation and its speed.

```
fluents:
car_position(C,D,S) requires car(C,_), segment(S,_,_,_,_), #int(D).
positive_orientation(C) requires car(C,_).
negative_orientation(C) requires car(C,_).
car_velocity(C,V) requires car(C,_), #int(V).
```

All fluents and actions will be related with cars because they are the only entities that could experiment a change. The actions we modeled are choosing segment, keeping speed and reducing speed.

```
actions:

choose_segment(S,C) requires car(C,_), segment(S,_,_,_,_).
keep_velocity(C,V) requires car(C,_), #int(V).
reduce_velocity(C,V) requires car(C,_), #int(V).
```

We also defined the changes on states. First we define the changes at the positions of the cars.

Let D1 be the position of a car, D the previous position and V its actual speed, if its orientation is positive D1=D+V, otherwise if its orientation is negative D1=D-V, in DLV-K we can not use operations like substraction so we use D=D1+V.

```
always:

caused car_position(C,D1,S) if car_velocity(C,V)
       after car_position(C,D,S), positive_orientation(C), D1=D+V.

caused car_position(C,D1,S) if car_velocity(C,V)
       after car_position(C,D,S), negative_orientation(C), D=D1+V.
```

Later, we eliminate previous states of position and orientation.

```
caused -car_position(C,D1,S) if car_position(C,D,S)
       after car_position(C,D1,S), D<> D1.

caused -positive_orientation(C) if negative_orientation(C)
       after positive_orientation(C).

caused -negative_orientation(C) if positive_orientation(C)
       after negative_orientation(C).
```

We defined changes of speed with the actions that affect it. Besides we have to eliminate the previous states.

```
caused car_velocity(C,V) after keep_velocity(C,V).
caused car_velocity(C,V) after reduce_velocity(C,V).
caused -car_velocity(C,V) if car_velocity(C,V1)
       after car_velocity(C,V), V<>V1.
```

We define restrictions for an action to be executable. -Car keeps maximum posible speed, that is, the minimum between the speed limit of the segment and top speed of the car. Moreover it is important to check that, when the car advances, the position of the car is not outside of the segment. Again we use two different rules, when the car has positive orientation or when it has negative orientation.

```
executable keep_velocity(C,V) if segment(S,_,_,L,Vs),
           car_position(C,D,S), carro(C,Vm), min(Vm,Vs,V),
           L>=R, R=D+V,  positive_orientation(C), #int(R).

executable keep_velocity(C,V) if segment(S,_,_,_,Vs),
           car_position(C,D,S), carro(C,Vm), min(Vm,Vs,V),
           D>=V, negative_orientation(C).
```

If the action is nonexecutable that is because the car is close to the end of the segment and therefore the car should reduce its speed, again the orientation is important. In this case the value of the new speed will be the remaining distance to the end of the segment, for that reason we verify that the remaining distance is smaller than the speed. If the orientation is positive the new speed will be the difference between length of the segment and the actual position of the car. Otherwise if the orientation is negative the speed will be the actual position of the car.

```
executable reduce_velocity(C,Vnew) if segment(S,_,_,L,_),
           car_position(C,D,S), car_velocity(C,V), L<R,
```

```
          R=D+V, positive_orientation(C), #int(R), L=Vnew+D,
          #int(Vnew),L<>D.

executable reduce_velocity(C,D) if  car_position(C,D,_),
          car_velocity(C,V), D<V, D<>0, negative_orientation(C).
```

When the car arrives to the end of the segment, it should choose another segment. The new position and the orientation of the car depend on if the car is at a start or end node. Besides, the actual position of the car is eliminated of the states.

```
executable choose_segment (S1,C) if car_position(C,L,S),
          segment(S,_,F,L,_), segment(S1,F,_,_,_),
          positive_orientation(C), S<>S1.

executable choose_segment (S1,C) if car_position(C,L,S),
          segment(S,_,F,L,_), segment(S1,_,F,_,_),
          positive_orientation(C), S<>S1.

caused car_position(C,0,S1) after choose_segment(S1,C),
       car_position(C,L,S), segment(S,_,F,L,_), segment(S1,F,_,_,_).

caused car_position(C,L1,S1) after choose_segment(S1,C),
       car_position(C,L,S), segment(S,_,F,L,_), segment(S1,_,F,L1,_).

caused negative_orientation(C) if car_position(C,L,S),
       segment(S,_,_,L,_) after choose_segment(S,C).

executable choose_segment (S1,C) if car_position(C,0,S),
          segment(S,I,_,_,_), negative_orientation (C),
          segment(S1,I,_,_,_), S<>S1.

executable choose_segment (S1,C) if car_position(C,0,S),
          segment(S,I,_,_,_), negative_orientation (C),
          segment(S1,_,I,_,_), S<>S1.

caused car_position(C,0,S1) after choose_segment(S1,C),
       car_position(C,0,S), segment(S,I,_,L,_),
       segment(S1,I,_,_,_).

caused car_position(C,L1,S1) after choose_segment(S1,C),
       car_position(C,0,S), segment(S,I,_,L,_),
       segment(S1,_,I,L1,_).

caused positive_orientation(C) if car_position(C,0,S)
       after choose_segment(S,C).

caused -car_position(C,D,S) after choose_segment(S1,C),
       car_position(C,D,S).
```

If the car is choosing a segment, it can not advance. If the car already choose a segment it should advance, it can not stay choosing segment.

```
nonexecutable keep_velocity (C,V) if choose_segment (S,C).
nonexecutable choose_segment(S,C) if choose_segment (S1,C), S<>S1.

%Inertial states

inertial car_position(C,D,S).
inertial positive_orientation(C).
inertial negative_orientation(C).

initially:

car_position(c1,0,a_b). positive_orientation(c1).
car_position(c2,9,b_d). negative_orientation(c2).


goal:

car_position(c1,10,b_d), car_position(c2, 0, c_b) ? (9)
```

4 Problems

The way DLV-K is structured was useful for modeling the movement of cars. DLV-K allows to generate states at different points in time. This stages together represent the movement of the cars. From the begin to the end, this language is more understandable than other descriptive languages oriented to logic programming.

In spite of this, the modeling of the problem was not so easy. We could not represent the whole problem. Although DLV-K has many advantages as a language for planning even under incomplete initial knowledge, after modeling this traffic problem we believe that it is not the best language to model it. In the original definition, cars do not have a general defined goal, this means that each car has its own goal.

The original objective is to simulate the interaction between cars. In DLV-K the simulation is only possible if there are well defined goals for each car. Even more, DLV-K groups in a general goal these individual goals. This causes that besides the expected plans, DLV-K generates undesarible models in the answer.

Consider this example:

We have two cars. The first one arrives to its goal in one step, and the second one needs three steps. The background knowledge defines two segments: a_b and b_c. The length of both the segments is 1. Initial position of c1 (the first car) is the node b and c2 (the second car) is at the node a.

We attempt to define that the cars arrived to their goals in the least number of steps. Also, we need that if a car arrives to its goal, it cannot continue causing troubles to the traffic of other cars. Moreover, the cars should not stay in a place in more than one step except when they are in their goals.

An expected model is:

```
STATE 0: positive_orientation(c1), positive_orientation(c2),
         car_position(c1,0,a_b), car_position(c2,0,b_c)
ACTIONS: keep_velocity(c1,1), keep_velocity(c2,1)
STATE 1: positive_orientation(c1), positive_orientation(c2),
         -car_position(c1,0,a_b), -car_position(c2,0,b_c),
         car_velocity(c1,1), car_velocity(c2,1),
         car_position(c1,1,a_b), car_position(c2,1,b_c)
ACTIONS: choose_segment(b_c,c1)
STATE 2: positive_orientation(c1), positive_orientation(c2),
         car_position(c2,1,b_c), -car_position(c1,1,a_b),
         car_position(c1,0,b_c)
ACTIONS: keep_velocity(c1,1)
STATE 3: positive_orientation(c1), positive_orientation(c2),
         car_velocity(c1,1), car_position(c1,1,b_c),
         car_position(c2,1,b_c), -car_position(c1,0,b_c)
PLAN:    keep_velocity(c1,1), keep_velocity(c2,1);
         choose_segment(b_c,c1); keep_velocity(c1,1)
```

But DLV-k gets also the next model:

```
STATE 0: positive_orientation(c1), positive_orientation(c2),
         car_position(car1,0,a_b), car_position(c2,0,b_c)
ACTIONS: keep_velocity(c1,1)
STATE 1: positive_orientation(c1), positive_orientation(c2),
         -car_position(c1,0,a_b), car_position(c2,0,b_c),
         car_velocity(car1,1), car_position(c1,1,a_b)
ACTIONS: choose_segment(b_c,c1)
STATE 2: positive_orientation(c1), positive_orientation(c2),
         car_position(c2,0,b_c), -car_position(c1,1,a_b),
         car_position(c1,0,b_c)
ACTIONS: keep_velocity(c1,1), keep_velocity(c2,1)
STATE 3: positive_orientation(c1), positive_orientation(c2),
         -car_position(c2,0,b_c), car_velocity(c1,1),
         car_velocity(c2,1), car_position(c1,1,b_c),
         car_position(c2,1,b_c), -car_position(c1,0,b_c)
PLAN:    keep_velocity(c1,1);  choose_segment(b_c,c1);
         keep_velocity(c1,1),  keep_velocity(c2,1)
```

According to the original definition this model is not right because the car c2 should advance and arrive to its goal in one step and it made it after 3 steps.

We also said in the original definition that two cars can be only in the same position at same time when they have opposite orientation.

At this previous model the model does not fit the definition -the car2 is not advancing at t0 -the car1 and car2 are in the same position at t1 and car2 is not advancing. -the car1 and car2 are advancing at the same time to the same node at t2.

Although we think that our modeling of the problem is natural and enough to obtain desirable answers, we realized that we would need another constraints to force the movement of a car before it arrives to its goal. That is, if a car has not carried out its goal it should continue advancing. Nevertheless, this is not easy because we would need to get the

goal of each car. Then, we should define twice the goals and this wouldn't be so natural.

Another problem, resulting of the one previously described is the interaction between the cars. As we could not force the movement of cars, it was possible that in a state, a car did not keep its velocity. This was a problem to model the velocity reduction because of the car that is in front. Into this rule we depend on the prediction of the next state for both cars and we are supposing the movement of the front car, but how we know this is not necesary true because some times the car does not do any action, so our rule for speed reduction will not work.

Aparently CCALC represent a best option for modeling this kind of problems. The modeling of the problem was more natural in CCALC because the language does not need a general defined goal. Also, its rule structure is more useful for handling constants and variables. We can make assigns for example:

```
caused speed(C)=min(Sp,Sp1) if nextsegment(C)=none
       & maxSpeed(C)=Sp & varsigmaAhead(C1,C)
       & -willLeave(C1) & speed(C1)=Sp1.
```

We do not believe these models are not interesting. Perhaps these models would be interesting for analisis in a different context. For example, we could think in a train station where somebody is trying to define the best route of each train and maybe the optimal plan is not necessarily the routes with least steps or without any stop for each train.

5 Conclusiones

The main problem we faced is that we have many agents that can modify the states. Each agent has its own initial and goal state, but they are similar in behavior. We have realized that DLV-K behaves better when we represent only one agent that interacts with static objects, but when we introduce more agents the modeling is not so intuitive.

DLV-K has many advantages as a language for planning even under incomplete initial knowledge. Maybe our implementation is not the most efficient, but it is the most natural.

References

[1] Thomas Eiterand Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. *Lecture Notes in Computer Science*, 1861:807–??, 2000.

[2] V.Akman, S. Erdogan, J. Lee, V. Lifschitz, and H. Turner. Representing the zoo world and the traffic world in the language of the causal calculator. Artificial Intelligence.