Going places - notes on a modular development of knowledge about travel

Michael Gelfond
Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
{mgelfond}@cs.ttu.edu

Abstract

The paper presents a formalization of a comparatively simple traveling story. The emphasis is on the development and implementation of a library of knowledge modules needed for axiomatization of journey - a movement of a group of objects from one place (the origin) to another (the destination). The movement should be able to follow a predefined route, and to achieve its goal even in the presence of natural interruptions, e.g. unexpected stops. We outline a language $\mathcal{M}$ for defining knowledge modules and for assembling them into a coherent knowledge base in CR-Prolog - an extension of Answer Set Prolog capable of encoding rare events. The formalization generalizes theory of action by introducing a notion of activity - a sequence of intended actions which can be interrupted by unexpected and unplanned events. The notion of journey presented in the paper is a special case of a more general notion of activity.

Introduction

The paper presents a formalization of a comparatively simple traveling story. The emphasis is on the development and implementation of a library knowledge modules needed for axiomatization of journey - a movement of a group of objects from one place (the origin) to another (the goal). The movement should be able to follow a predefined route, and to achieve its goal even in the presence of natural interruptions, e.g. unexpected stops. For simplicity we will be mainly, interested in whereabouts of various objects at different stages of the journey. The following example will be used to illustrate the proposed methodology.

Example
Consider the following story: *John and Mary decided to go on a trip from El Paso to Dallas. On the way they planned to stop in Lubbock to pick up Bill.*
1. What is the planned trajectory of the participants?
2. Where do we expect them to be after the trip?
3. Would they visit[1] *Carlsbad?*

Since people normally follow their plans, it is safe to assume that the pair would leave El Paso and follow their planned trajectory: El Paso, *en route*, Lubbock, *en route*, Dallas. Bill will be in Lubbock at the time of his friends arrival and then change his position (first to *en route* and next to Dallas). The answer to the last two questions are therefore *in Dallas* and *no*.

Suppose now that *after departure from El Paso Mary and John made an unplanned stop in Carlsbad to see the caverns*.

Despite this unexpected event we assume that the original intention persists, and hence after the unplanned stop the journey will continue as planned. The new trajectory for the pair is El Paso, *en route*, Carlsbad, *en route*, Lubbock, *en route*, Dallas with Bill joining at the last leg of the journey. The answer to second question is unchanged while that to the third one is now *yes*.

To obtain these answers we need a logical language capable of reasoning about defaults and (unexpected) exceptions to defaults, as well as reasoning about intentions, and effects of actions. Our language of choice is CR-Prolog (Balduccini & Gelfond 2003b) which is an extension of Answer Set Prolog (ASP) (Gelfond & Lifschitz 1991) - the language of logic programs with two negations and disjunction under the answer set semantics. We believe CR-Prolog to be superior to the original ASP because of its ability of dealing gracefully with unexpected exceptions to defaults. We assume that the reader is familiar with the syntax and semantics of ASP. For basic definitions and a comprehensive account of its use for knowledge representation one may look at (Baral 2003). A brief account of CR-Prolog will be given in the next section.

CR-Prolog

A program of (a relevant subset of) CR-Prolog is a pair consisting of a signature and a collection of rules of the form:

$$l_0 \leftarrow l_1, \ldots, l_n, \text{ not } l_{n+1}, \ldots, \text{ not } l_m \qquad (1)$$

and

$$r : l_0 \text{ +- } l_1, \ldots, l_n, \text{ not } l_{n+1}, \ldots, \text{ not } l_m \qquad (2)$$

where $l_1, \ldots, l_n$ are literals, and r is a term representing the name of the rule. Rules of type (1) and (2) are called *regular* and *consistency restoring* rules (cr-rules) respectively. The set of regular rules of cr-program Π will be denoted by Π^r; the set of cr-rules of Π will be denoted by Π^{cr}. By $\alpha(r)$ we

[1] We understand visit as "spending time at a place with a certain intent". Hence one can pass through a town without visiting it.

denote a regular rule obtained from a consistency restoring rule r by replacing +- by $\leftarrow$; α is expanded in a standard way to a set R of cr-rules.

A minimal (with respect to set theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π.

A set A is called an *answer set* of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

Example: Consider a program P

```
p(X)  :- not ab(X).
ab(1).
q(2).
s(X) :- p(X),q(X).
r(X) : ab(X) +-.
```

The program includes a default with an exception, 1, a partial definition of s in terms of p and q, and consistency restoring rule which acknowledges the possible existence of unknown exceptions to the default. Since normally such a possibility is ignored the answer set of the program consists of its facts and atoms $p(2), s(2)$.

Suppose now that the program is expanded by a new atom, -$s(2)$. The regular part of the new program has no answer set. The cr-rule solves the problem by assuming that 2 is a previously unknown exception to the default. The resulting answer set consists of the program facts and the atom $ab(2)$. The experimental system, *crmodels* (Kolvekal 2004), computes answer sets of programs of CR-Prolog. It can be downloaded from

```
http://krlab.cs.ttu.edu/~marcy/crmodels/
```

It is also useful to download the output formatting program *mkatoms* from

```
http://krlab.cs.ttu.edu/~marcy/mkatoms/
```

To run a program of CR-Prolog one can store it in a file, say, F and type the command line

```
crmodels F | mkatoms
```

Even though the above program P is syntactically and semantically correct, *crmodels* will, unfortunately, be unable to accept it as a correct input. To make it work one needs to either ground the variables in the cr-rule replacing it by

```
r(1) : ab(1) +-.
r(2) : ab(2) +-.
```

or use the input language of Lparse - a version of ASP implemented in an answer set solver Smodels (Niemela & Simons 2000). In particular we are required to define the types of variables used in the names of cr-rules. In our case this can be done by statements

```
const n = 2.
object(1..n).
#domain object(X).
```

which guarantee that the variable X takes on values from the set $\{1, \ldots, n\}$ with $n = 2$.

Building the Library (*action* and *atomic_move*)

We start with the development of several library modules which may be used to describe behavior of agents in dynamic domains. The underlying model of such a domain is given by a transition diagram. Its states are sets of fluents, i.e. (possibly partial) functions of time-steps, and its arcs are labeled by actions. Paths of the diagram correspond to all possible trajectories of the dynamic domain. We briefly outline a language, $\mathcal{M}$, for defining knowledge modules and assembling them into a coherent knowledge base. Work on design and implementation of $\mathcal{M}$ is in its initial stage. The language is mentioned primarily in a hope to stress the need for such languages and to elicit comments. Our long term goal is to build a library of classes written in $\mathcal{M}$ and organized into a hierarchy.

Syntactically class definition of $\mathcal{M}$ is of the form:

```
class class_name : type
body
```

The first line indicates the position of the class in the class hierarchy. The *body* consists of two parts: the class' signature and the class' axioms. As our first example we define a class **action** which contains the ASP representation of the inertia axiom (Hayes & McCarthy 1969) and the basic axioms for intended actions.

Action

```
class action : top
signature
sorts: step,action,fluent,range(fluent).
holds(range(fluent), fluent, step).
occurs(self, step).
intend(self, step).
defined(fluent,step).
start, end : self --> step.
next(step,step).
```

The names *step*, *action*, *fluent* and *range* of uninterpreted sorts are followed by the list of relations used by the axioms of the class. Sorts of parameters of a predicate symbol are listed in parentheses. Every time an instance a of class *action* is created *self* will be replaced by a. Intuitively, $holds(y, f(x), i)$ says that at step i, fluent $f(x)$ has value y; statement $occurs(a, i)$ says that action a was executed at step i; $intend(a, i)$ states that at step i the agent came to a decision to execute action a. (Note that in accordance with the theory of intentions from (Baral & Gelfond 2005) this implies that a will be actually executed as soon as possible). Statement $defined(f, i)$ holds if fluent f is defined at step i; $next(i1, i2)$ indicates that $i1$ immediately follows $i2$.

The second part of the class definition starts with the word *axioms*, and is followed by the list of axioms of the class.

```
axioms
#domain step(I;I1;I2).
#domain fluent(F;F1;F2).
#domain action(A;A1;A2).
```

The *domain* statements above define sorts of variables used in the axioms of the class. (As mentioned before this syntax is borrowed from Smodels and is understood by CR-Prolog). The *domain* axioms are needed only to make the resulting program runnable on a grounding program Lparse, and hence on CR-Prolog. It is worth noting that the necessary typing information is present in the signature part of the definition and can be, in the future, used by a grounding program incorporated in $\mathcal{M}$. This may lead to a better grounding schema which makes these axioms unnecessary. The details of this will be discussed in a future paper on $\mathcal{M}$. We also have the *inertia axiom* which says that things normally stay as they are:

```
holds(Y,F,I1) :-
        next(I1,I),
        range(Y,F),
        defined(F,I1),
        holds(Y,F,I),
        not -holds(Y,F,I1).
```

Note that $range(Y, F)$ and $defined(F, I1)$ guarantee that Y belongs to the range of fluent F which is defined on $I1$. Next we include basic axioms for *intended actions* (Baral & Gelfond 2005):

```
occurs(A,I) :-
        intend(A,I),
        not -occurs(A,I).
intend(A,I1) :-
        next(I1,I),
        intend(A,I),
        -occurs(A,I),
        not -intend(A,I1).
```

The first axiom says that normally the agent does not procrastinate in acting on his intentions. The second ensures that unrealized intentions persist. We also need auxiliary axioms:

```
 -holds(Y1,F,I) :-
        range(Y1,F),
        range(Y2,F),
        holds(Y2,F,I),
        Y1 != Y2.

start(A,I) :-
       occurs(A,I).
end(A,I+1) :-
       occurs(A,I).
```

and

```
-occurs(A,I2) :-
       occurs(A,I1),
       neq(I1,I2).
```

which state that fluents are functions, that execution of actions take exactly one step, and that action names stand for particular occurrences of actions. The last requirement should be contrasted to the treatment of names in many other action formalisms, which allow histories of the form $occurs(a, 1), occurs(a, 2)$. Here a names an action class whose instances are specified by the second parameter of $occurs$. In our formalism such a history will be inconsistent. We believe that the disadvantage of the necessity to introduce multiple names will be offset by the comparative simplicity in dealing with sequences of actions.

This class can be easily extended by, say, allowing non-inertial fluents, continuous as well as discrete processes, etc. But we hope that this comparatively simple class is sufficient to illustrate our methodology.

There are three operations which can be applied to a class C of $\mathcal{M}$. It can be

1. refined to define a new, more specific class, C_0;
2. inherited by some other class, C_1;
3. used to create a particular instance, c of C.

To illustrate the first two operations we use the class *action* to define a new class, *atomic move*.

Atomic Move

An action *atomic move* changes locations of its participants from the action's origin to its destination. Instances of *atomic move* may be supplied with a set of possible locations, the destination and the origin of the move, and the moves participants. Here is the definition:

```
class atomic_move : action
signature
 sorts: location, object.
 origin, dest : self --> location.
 participant(object,self).
 fluent loc : self+object --> location.
```

The first line ensures that class *atomic move* inherits the signature and the axioms of class *action*. It has additional sorts, $location$ and $object$. Functions $origin$ and $dest$ map the move into the corresponding locations. Relation $participant$ lists the objects (people and things) involved in the move; $loc(x)$ is a fluent which gives the position of a participant x before and after the move. We also find it convenient to define fluent $loc(m)$ for atomic move m. The fluent is defined at two time points - the start and the end of the execution of m. The atomic move, m, changes its position from the origin to the destination.

Now we define the axioms of *atomic_move*.

```
axioms
#domain object(O;O1;O2).
#domain location(L;L1;L2).
#domain atomic_move(M;M1;M2).
```

The first three statements define the new sorted variables. Axiom

```
action(M).
```

says that atomic moves are actions. Axioms

```
fluent(loc(O)).  range(L,loc(O)).
fluent(loc(M)).  range(L,loc(M)).
```

define the corresponding fluents and their ranges. Axioms

```
defined(loc(M),I) :-
          start(M,I).
defined(loc(M),I) :-
          end(M,I).
-defined(loc(M),I) :-
          not defined(loc(M),I).
defined(loc(O),I).
```

define the domains of fluents. They state that fluent $loc(M)$ is defined at the beginning and the end of the execution of move, while fluent $loc(O)$ is defined at any step of a possible trajectory of the domain. Note that these axioms populate sorts *fluent*, *range(fluent)* and *action* which were left uninterpreted in the class *action*.

The next statement represents logic programming encoding of dynamic causal law for an atomic move. Recall that in action languages (see for instance (Gelfond & Lifschitz 1998) dynamic causal law is a statement of the form

$$a \text{ causes } f \text{ if } p \tag{3}$$

which says that if a were executed in a state satisfying conditions from p then f will be true in a successor state of a. The statement

```
holds(Y,loc(M),I1) :-
        next(I1,I),
        range(Y,loc(M)),
        occurs(M,I),
        dest(Y,M).
```

is a direct translation of the corresponding causal law which says that the atomic move is always successful, i.e. moves m to its destination.

Locations of the participants of an atomic move m are determined by the location of m. In action languages this is expressed by a state constraint - statement of the form

$$f \text{ if } p \tag{4}$$

which says that every state of the domain satisfying conditions from p also satisfies f. The constraint determining locations of objects has the form:
$loc(O) = Y$ if $participant(O, M), loc(M) = Y$.
The logic programming translation of this constraint looks as follows:

```
holds(Y,loc(O),I) :-
        range(Y,loc(M)),
        participant(O,M),
        holds(Y,loc(M),I).
```

The next statement represents an executability condition for atomic move M. It says that one cannot move to its current location.

```
-occurs(M,I) :-
    dest(L,M),
    holds(L,loc(M),I).
```

We also need the collection of *initialization* axioms. These axioms are not needed for the description of successor states of *atomic move*. Their role is to ensure that a given scenario (collection of observations and occurrences of actions at particular time steps) defines correct collection of trajectories of the diagram.

```
1 {holds(X,loc(O),1) : location(X)} 1.
holds(L,loc(M),I) :-
    occurs(M,I),
    origin(L,M).
moved(O,I) :-
     participant(O,M),
     occurs(M,I).
:- next(I1,I),
   participant(O,M),
   holds(L1,loc(O),I),
   holds(L2,loc(O),I1),
   L1 != L2,
   not moved(O,I).
-holds(L,loc(M),I) :-
    occurs(M,I),
    dest(L,M).
```

The first initialization axiom is written in the input language of Lparse. Informally its ground instance for object o can be viewed as a shorthand for

(i) a disjunction

$holds(l_1, loc(o), 1)$ or $\ldots$ or $holds(l_n, loc(o), 1)$

where $\{l_1, \ldots, l_n\}$ is the range of $loc(o)$;

(ii) constraints

:- $holds(l_i, loc(o), 1), holds(l_j, loc(o), 1)$

for every $l_i \neq l_j$.

The axioms ensures that in the initial state every object of the domain has a unique position. Of course if this position is unknown the formalization may (correctly) have multiple models. To understand the need for the second initialization axiom consider a scenario represented by the statement $occurs(a, 1)$ where a is an atomic move with a participant o and origin l. Without the connection between the origin of the move and positions of its participant the program will have multiple models containing $holds(l_i, loc(o), 1)$ for every possible initial location l_i of o. The connection established by the second axiom eliminates all the models except the one containing $holds(l, loc(o), 1)$, which corresponds to the correct trajectory. The need for the next two axioms is illustrated by the scenario $holds(l_1, loc(o), 1),\ occurs(a, 2)$. It is easy to check that, since $holds(l, loc(o), 2)$ derived by the second initialization axiom defeats the inertia, the program will have an answer set containing a "non-existent" trajectory in which o miraculously changes its position from l_1 to l. The third and fourth initialization axioms eliminate this possibility[2].

Formalizing the Story

The library consisting of classes *action* and *atomic_move* can be used to create instances of actions, and knowledge bases

[2] Admittedly this solution of the problem seems to be overly complex. There is a simpler and more elegant solution but it requires extension of the language of action theories by new predicate symbols for observations in the style of (Balduccini & Gelfond 2003a). We plan to investigate this approach in our future work.

containing information about these instances. To illustrate this let us consider a simplified description of the first leg of the journey described in our main story: *John and Mary went from El Paso to Lubbock*. The statement describes an action *move* with clearly specified origin, destination, and participants. Here is a logical representation, P_1, of this story in $\mathcal{M}$:

```
knowledge_base P1
include 'library'
instance a : atomic_move
   origin = el_paso
   dest = lubbock
   participant(mary)
   participant(john).
regular part
const n = 2.
step(1..n).
next(I+1,I) :- I <  n.
occurs(a,1).
```

A knowledge base of $\mathcal{M}$ consists of three parts: the *library* of classes, the set of *instances* of classes from this library, and the *regular part* - collection of rules of CR-Prolog. We assume that the library of P_1 contains classes *action* and *atomic_move*. Instance a is defined as an instance of atomic_move by Mary and John from El Paso to Lubbock. Note that for simplicity statements in the body of the definition of a omit the second parameter a of the functions, *origin* and *destination* and predicate symbol *participant*. The first three statements of the regular part of P_1 (written in the input language of Lparse) define two consecutive steps, 1 and 2. The last sentence says that instance a of atomic_move occurs at step 1. A not yet existent implementation of $\mathcal{M}$ will take a knowledge base P_1, extract definitions of *atomic move* and *action* from the library, and create a logic program, $\mathcal{M}(P_1)$ consisting of:

1. Direct translation of the body of instance a into the language of *atomic move*:

```
atomic_move(a).
origin(el_paso,a).
dest(lubbock,a).
participant(mary,a).
participant(john,a).
```

2. The list of atoms describing (previously uninterpreted) sorts *location* and *object* of *atomic move:*

```
location(lubbock).
location(el_paso).
object(mary).
object(john).
```

3. Axioms from classes *action* and *atomic move*.

4. Regular rules of P_1.

Note that the information necessary for forming the lists from (1) and (2) can be extracted from P_1 and the signature parts of classes *action* and *atomic move*. The resulting program contains no uninstantiated sort.

To display the locations of participants before and after the move we may use the rules:

```
loc(O,Y,I) :-
    range(Y,loc(O)),
    holds(Y,loc(O),I).
hide. show loc(O,Y,I).
```

To run the program one can store it in a file, say, F and type

```
crmodels F | mkatoms
```

The program will display

```
loc(john,el_paso,1)
loc(john,lubbock,2)
loc(mary,el_paso,1)
loc(mary,lubbock,2)
```

To better understand our formalization of intended actions from the class *action* it may be instructive to look at programs Q_1 and Q_2. Q_1 is obtained by replacing the rule $occurs(a,1)$ of P_1 by

```
intend(a,1).
```

Program Q_2 is obtained by replacing $const\ n = 2$ and $occurs(a,1)$ of P_1 by

```
const n = 3.
intend(a,1).
-occurs(a,1).
```

To display occurrences of a we add rules:

```
o(A,I) :-
    occurs(A,I).
show o(A,B).
```

Note that the output of Q_1 is $o(a,1)$ - the intended action occurred immediately. The output of Q_2 is $o(a,2)$ - intention to perform a which could not be realized at 1 persisted, and a occurred at 2.

Building the Library (*Sequences*)

In this section we introduce a few more simple classes which will be used in our formalization of the traveling story.

In addition to single actions we often need to represent sequences of actions, and even sequences of such sequences, etc. The standard list operator [ ] of classical Prolog allows simple and elegant representation of such objects. This representation however leads to infinite Herbrand universe of the program which rules out reasoning with current answer set solvers. As a result sequence $s = \langle s_1, \ldots, s_k \rangle$ in $\mathcal{M}$ will be represented by the collection of atoms of the form

```
length(k,s).
index(1..k).
component(sj,j,s)
```

where $1 \leq j \leq k$. (We will of course assume that components s_j's of s are defined before the s.) Now we are ready to expand our library by a new class, *sequence* of basic elements, sequences of basic elements, etc.

Sequence

```
class sequence(element)
signature
sorts : index
component(sequence+element,index,sequence)
length : self --> index
axioms
#domain index(K;K1;K2).
sequence(X) :-
        component(Y,K,X).
```

The class defines sequences constructed from the basic elements, i.e. instances of the class *element*. If $a_1, a_2,$ and a_3 are basic elements, then sequences $s_1 = \langle a_2, a_3 \rangle$, $s_2 = \langle a_1, s_1 \rangle$, etc. are instances of the class *sequence(element)*. To create an instance of this class one has to have a library class *element* whose axioms will be added to the axioms of *sequence(element)*. It maybe useful to expand the latter axioms by defining various operations on sequences but we will not discuss this possibility here. The next class will be useful for formalizing our travel story which can be seen as a sequence of actions.

Action Sequence

The class *action sequence* is built by setting the parameter *element* of the class *sequence* to *action* and by adding additional axioms defining the execution of an action sequence and the notion of its intended execution.

```
class action_sequence : sequence(action)
signature
occurs(action_sequence,step)
intend(action_sequence,step)
start, end : action_sequence --> step
axioms
#domain action_sequence(S;S1;S2).
sequence(S).
element(A).
element(S).
#domain element(V;V1;V2).
```

Axioms

```
occurs(V,I) :-
     occurs(S,I),
     component(V,1,S).
occurs(V2,I1) :-
     occurs(S,I),
     component(V2,K+1,S),
     component(V1,K,S),
     end(V1,I1).
```

say that execution of an action sequence S at step I implies execution at I of its first component, and that execution of the $(K+1)$'th component of S starts immediately after the end of the execution of its K'th component. Similarly, the next two axioms propagate intentions from execution of sequences to executions of its component.

```
intend(V,I) :-
     intend(S,I),
     component(V,1,S).
intend(V2,I1) :-
     intend(S,I),
     component(V2,K+1,S),
     component(V1,K,S),
     end(V1,I1).
```

Finally we define beginning and end of the execution of a sequence S and the step when the intent to execute S was formed.

```
start(S,I) :-
     occurs(S,I).
end(S,I) :-
     length(N,S),
     component(V,N,S),
     end(V,I).
start_intent(S,I) :-
     intend(S,I).
```

Formalizing the Story (continued)

Now we show how to use the newly created library to formalize the first part of our traveling story. We will do it by creating an instance of action sequence, m, consisting of two consecutive moves, a_1 and a_2.

```
knowledge_base P2
include 'library'
instance a1 : atomic_move
   origin = el_paso
   dest = lubbock
   participant(mary).
   participant(john).
instance a2 : atomic_move
   origin = lubbock
   dest = dallas
   participant(mary).
   participant(john).
   participant(bob).
instance m : action_sequence
   const k = 2.
   index(1..k).
   component(a1,1,m).
   component(a2,2,m).
regular part
   const n = 2.
   step(1..n).
   next(I+1,I) :- I <  n.
   occurs(m,1).
```

The program $\mathcal{M}(P_2)$ consists of axioms from the library classes and collection of facts

```
% instance a2 : atomic_move
atomic_move(a1).
   origin(el_paso,a1).
   dest(lubbock,a1).
   participant(mary,a1).
   participant(john,a1).
% instance a2 : atomic_move
atomic_move(a2).
   origin(lubbock,a2).
   dest(dallas,a2).
   participant(mary,a2).
   participant(john,a2).
```

```
   participant(bob,a2).
% instance m : action_sequence
action_sequence(m).
   const k = 2.
   index(1..k).
   length(k,m).
   component(a1,1,m).
   component(a2,2,m).
% Sorts extracted from instances
location(lubbock).
location(el_paso).
location(dallas).
object(mary).
object(john).
object(bob).
% Regular part
const n = 3.
step(1..n).
next(I+1,I) :- I <  n.
intend(m,1).
```

The output of this program is

```
loc(bob,lubbock,1)
loc(john,el_paso,1)
loc(mary,el_paso,1)
o(a1,1)
loc(bob,lubbock,2)
loc(john,lubbock,2)
loc(mary,lubbock,2)
o(a2,2)
loc(bob,dallas,3)
loc(john,dallas,3)
loc(mary,dallas,3)
```

which is the trajectory described by the first part of our traveling story. It is not clear however how to elegantly expand this formalization by the information from the second part of the story. Our theory of intention for action sequences does not appear to be sufficiently elaboration tolerant to allow the intervention of unplanned actions. The class *activity* introduced in the next section will be used to remedy the situation.

Building the Library (*Activity*)

A new class, *activity*, is a specialization of the class *action_sequence*. It allows for unplanned, unexpected actions. The signature of *activity* contains two types of actions: *planned* and *possible*. Actions used in relation $component(a, k, s)$ inherited from *action_sequence* are planned. The statement is now read as "a is the k-th *planned* action of activity s". A new relation, $possible\_act(a, s)$ says that a is a possible unplanned action which can be unexpectedly executed by (participants of) s. This would allow us to represent planned trip El Paso, Lubbock, Dallas as well as an unexpected visit to Carlsbad. A relation $act\_of(a, s)$ is satisfied by the union of planned and possible actions of s.

Activity

```
class activity : action_sequence
signature
possible_act(action,self)
act_of(action,self)

axioms
action_sequence(X) :-
       activity(X).
planned_act(A,S) :-
       component(A,K,S).
planned_act(A,S2) :-
       component(S1,K,S2),
       planned_act(A,S1).
act_of(A,J) :-
   planned_act(A,J).
act_of(A,J) :-
   possible_act(A,J).
```

The first five axioms define two of the new relations. The information about possible actions will be supplied by the definition of the corresponding activity instance. The next axiom is a substantial addition to the formalization of intentions contained in *action sequence*. It says that *no intended action remains at the end of the activity*.

```
:- intend(A,n).
```

To satisfy this constraint the activity may use its possible actions. This is expressed by the only consistency-restoring rule of this formalization:

```
r(A,I):occurs(A,I) +- possible_act(A,S).
```

In the next section we use the new class to formalize the notion of *journey*.

Building the Library (*Journey*)

A *journey* or a *trip* is an activity which is allowed to have many participants who may join and leave it at any reasonable step i. The journey may follow complex routes and stop in unplanned places. To formalize the notion of journey we first expand our library by new classes: *depart arrive*, *embark* and *disembark*. We start with defining *depart* and *arrive* which can be viewed as specializations of *atomic_move*.

Depart

```
class depart : atomic_move
axioms
  #domain depart(D;D1;D2).
  atomic_move(D).
  dest(en_route,D).
```

Arrive

```
class arrive : atomic_move
axioms
  #domain arrive(Arr;Arr1;Arr2).
  atomic_move(Arr).
  origin(en_route,Arr).
  -occurs(Arr,I) :-
     -holds(en_route,loc(Arr),I).
```

Now a *simple move* m from location l_1 to location l_2 can be represented by an action sequence which consists of departing l_1 and arriving at l_2. The intent of performing m will, under normal circumstances lead to the corresponding

departure and arrival. By expanding m with a set of possible actions we can turn it into activity. Now, in contrast to atomic move, m can be interrupted by, say, unexpected stop at some location l_3. In this case we expect the intention to persist and the trip to depart l_3 and to arrive at l_2.

Our journey will normally start with action *embark* and end with action *disembark*. (Of course some participants will be able to embark and disembark while the journey is in progress). The following terminology will be useful for defining *embark* and *disembark*: if e is an action of, say, John and Mary embarking on a trip m then John and Mary are *participants* of e and m is its *target*. We will also need a fluent $is\_participant(o, m)$ which holds at step i if, at i, object o is a participant of trip m. Clearly, execution of e causes the fluent $is\_participant(john, m)$ to become true.

Embark

```
class embark : action
target(self,activity)
participant(object,self)
fluent is_participant :
       object * self --> boolean
```

Note that the (not yet developed) implementation of $\mathcal{M}$ will make sure that *embark* inherits axioms of *activity*.

```
axioms
#domain embark(E;E1;E2).
action(E).
fluent(is_participant(O,V)).
range(true,is_participant(O,V)).
range(false,is_participant(O,V)).
```

The above mentioned causal law has the form

```
holds(true,is_participant(O,V),I1) :-
         next(I1,I),
         occurs(E,I),
         target(V,E),
         participant(O,E).
```

The time span of the new fluent is given by

```
defined(is_participant(O,V),I) :-
         start(V,I1),
         end(V,I2),
         I1 <= I, I <= I2.
-defined(is_participant(O,V),I) :-
     not defined(is_participant(O,V),I).
```

We also have two executability conditions:

```
-occurs(E,I) :-
    target(V,E),
    holds(en_route,loc(V),I).
 -occurs(E,I1) :-
    target(V,E),
    next(I2,I1),
    holds(L1,loc(V),I2),
    participant(O,E),
    holds(L2,loc(O),I1),
    L1 != L2.
```

To save space we'll omit a similar definition of *disembark*.

Journey

By a *journey* we mean an activity consisting of atomic moves *depart* and *arrive*, and actions *embark* and *disembark*. For simplicity assume that

1. at step i of the journey no actions are performed at different locations;
2. atomic moves of the journey have no explicitly specified participants. Participation in the journey is fully controlled by actions *embark* and *disembark*.
3. During the life of a journey at least one of its actions is performed at each time-step.

```
class journey : activity
axioms
#domain journey(J;J1;J2).
activity(J).
```

We start with axioms defining a fluent $loc(J)$.

```
fluent(loc(J)).
range(L,loc(J)).
defined(loc(J),I) :-
   start(J,I1),
   end(J,I2),
   I1 <= I, I <= I2.
-defined(loc(J),I) :-
   not defined(loc(J),I).
```

The next axiom is a state constraint which defines the location of a journey in terms of locations of actions which happen during its life span (Assumption 2 above insures consistency of this definition).

```
holds(Y,loc(J),I) :-
    act_of(A,J),
    range(Y,loc(A)),
    holds(Y,loc(A),I).
```

The next axiom relates locations of two consecutive actions.

```
holds(Y,loc(A2),In) :-
    next(In,I),
    range(Y,loc(A1)),
    range(Y,loc(A2)),
    start(J,I1),
    end(J,I2),
    I1 <= I, I < I2,
    occurs(A1,I),
    occurs(A2,In),
    holds(Y,loc(A1),In).
```

It says that if an activity action A_1 ends at location Y then the next activity action, A_2, starts at Y. Next we have the state constraint which says that participants share their location with the journey,

```
holds(Y,loc(O),I) :-
    range(Y,loc(J)),
    holds(true,is_participant(O,J),I),
    holds(Y,loc(J),I).
```

and rules representing constraints (1) - (3) we imposed on our journeys.

Constraint 1:

```
:- act_of(X1,J),
   act_of(X2,J),
   occurs(X1,I),
   occurs(X2,I),
   holds(L1,loc(X1),I),
   holds(L2,loc(X2),I),
   neq(L1,L2).
```

Constraint 2:

```
:- act_of(M,J),
   participant(O,M).
```

Constraint 3:

```
something_happend(I) :-
          occurs(A,I).
:- act_of(X1,J),
   act_of(X2,J),
   occurs(X1,I),
   next(I1,I),
   next(I2,I1),
   occurs(X2,I2),
   not something_happend(I1).
```

Formalizing the story

Now we show how the class *journey* can be used to represent our original traveling story.

```
knowledge_base Q
include 'library'
```

We start with creating instances of relevant actions.

```
instance d(L) : depart
   origin = L
instance a(L) : arrive
   dest = L
instance m : journey
   const k = 6
   index(1..6)
   component(e1,1,m).
   component(d(el_paso),2,m)
   component(a(lubbock),3,m)
   component(e2,4,m)
   component(d(lubbock),5,m)
   component(a(dallas),6,m)
   possible_act(a(L),m)
   possible_act(d(L),m)
instance e1 : embark
   target(m,e1).
   participant(mary,e1)
   participant(john,e1)
instance e2 : embark
   target(m,e2)
   participant(bob,e2)
```

Running this program together with a regular part

```
const n = 9.
intend(m,1).
```

will output the first intended trajectory. To save space we only show the output for Mary and Bob.

```
loc(bob,lubbock,1) loc(mary,el_paso,1)
o(e1,1)
loc(mary,el_paso,2) loc(bob,lubbock,2)
o(d(el_paso),2)
loc(mary,en_route,3) loc(bob,lubbock,3)
o(a(lubbock),3)
loc(mary,lubbock,4) loc(bob,lubbock,4)
o(e2,4)
loc(mary,lubbock,5) loc(bob,lubbock,5)
o(d(lubbock),5)
loc(mary,en_route,6) loc(bob,en_route,6)
o(a(dallas),6)
loc(mary,dallas,7) loc(bob,dallas,7)
```

The unintended stop in Carlsbad can be represented by simply expanding the regular part by

```
occurs(a(carlsbad),3).
```

Now the output will be

```
loc(bob,lubbock,1) loc(mary,el_paso,1)
o(e1,1)
loc(mary,el_paso,2) loc(bob,lubbock,2)
o(d(el_paso),2)
loc(mary,en_route,3) loc(bob,lubbock,3)
o(a(carlsbad),3)
loc(mary,carlsbad,4) loc(bob,lubbock,4)
o(d(carlsbad),4)
loc(mary,en_route,5) loc(bob,lubbock,5)
o(a(lubbock),5)
loc(mary,lubbock,6) loc(bob,lubbock,6)
o(e2,6)
loc(mary,lubbock,7) loc(bob,lubbock,7)
o(d(lubbock),7)
loc(mary,en_route,8) loc(bob,en_route,8)
o(a(dallas),8)
loc(mary,dallas,9) loc(bob,dallas,9)
```

Note that the unexpected stop in Carlsbad happened when the journey, m, was intending to arrive in Lubbock. After finding itself on the ground in Carlsbad it "tries" to perform the next intended action $a(lubbock)$. Since arrival is executable only if the trip is *en_route* this cannot be done. Hence the trip first performs the action $d(carlsbad)$ "found" by the consistency restoring rule of the class *activity*.

The computation performed by the program takes a considerable amount of time. The example above took almost a minute to run. The performance can be substantially improved by improving the efficiency of *crmodels*. (There is some ongoing work in this direction). Another source of inefficiency is the size of a grounding program. We hope that this can be reduced by supplying $\mathcal{M}$ with a smart grounding mechanism independent of Lparse. Note also that if program Q were written in a fully developed and implemented $\mathcal{M}$ the sorts, e.g. *location* and *object* would be computed automatically. More importantly the program would be able

to save the writer a substantial amount of time by warning him about a number of errors caused by misspelled names, missing parameters, change of parameters' order, etc.

Discussion

In this paper

1. We introduced an ASP based theory, say, $\mathcal{A}$ of activity - a sequence of intended actions together with a collection of possible events which can interfere with the agent's intentions. $\mathcal{A}$ includes ASP axioms describing the effects of actions and action sequences. In addition it contains a new theory of intentions which expands the one described in (Baral & Gelfond 2005). $\mathcal{A}$ allows us to reason about the behavior of agent(s) performing an intended activity (even in the presence of possible unexpected interruptions).
2. We used the theory of activity to formalize the notion of a journey, i.e. the act of traveling from one place to another. A journey normally has many participants who may join and/or leave it at any reasonable step i. It may follow complex routes and stop in unplanned places.
3. We used the example of building axioms of *journey* to illustrate our methodology of using ASP for formalizing common-sense knowledge.
4. The axioms were used to formalize a simple example of a journey with an unexpected stop.
5. We also outlined the basic idea of the language $\mathcal{M}$ for defining knowledge modules and assembling them into a coherent knowledge base. Since the language is in the beginning stages of its development our description is rather informal. We hope however that it may give some useful insight into the basic features of the language.

The axioms of the class *journey* have grown from those of the travel module presented in (Baral, Gelfond, & Scherl 2004). The structure of formalization however is substantially improved and generalized. It is based on a simple, but hopefully useful, notion of activity[3]. The use of CR-Prolog for formulating the theory of intentions is new. It allows the journey to resume attempts to achieve its intended goals after the unexpected interruptions.

A number of formal languages and techniques were used in the past to accurately formalize various traveling stories. (see for instance (Lifschitz 2000), (Mueller 2004) among others). The former uses the formalism from (Giunchiglia *et al.* 2004) while the later is based on Circumscriptive Event Calculus from (Shanahan 1995). In these and other cases the emphasis and the techniques used were substantially different from those presented in this paper. There is also a number of proposals for extensions of the ASP Prolog and/or action languages which provide means for modular development of larger programs. many of them address problems similar to those confronted by $\mathcal{M}$. The language, DLT, suggested in (Calimeri *et al.* 2004) expands ASP with templates. The existing implementation is built on top of the powerful answer set solver DLV (Calimeri *et al.* 2002). A template

```
#template max[p(1)](1) {
exceeded(X) :- p(X), p(Y), Y > X.
max(X) :- p(X), not exceeded(X).
}
```

of DLT is defining "the predicate *max*, intended to compute the maximum value over the domain of a generic predicate p". The next two rules

```
:- max[w(*)](X), X > 100.
:- max[v(*)](X), X > 50.
```

say that no number satisfying property w can be greater than 100 and no number satisfying property v can be greater than 100. A sophisticated matching algorithm translates a program P_1 in DLT into the corresponding ASP program P_1. The part of $\mathcal{M}$ presented in this paper seems to address problems somewhat orthogonal to those dealt with in DLT. We are more interested in providing a simple mechanism for organizing modules (especially those needed for reasoning about dynamic domains) into a hierarchy, and in using this hierarchy for building knowledge bases. It seems that the reuse of rules similar to those defined in the DLT template *max* can be achieved by reifying properties of the original language. A macro, say,

```
#reify(p)
```

occurring in a knowledge base will be replaced by

```
is_a(X,p) :- p(X)
```

The general definition of *max* will be given by rules

```
exceeded(X,S) :-
       is_a(X,S),
       is_a(Y,S),
       Y > X.
max(X,S) :-
       is_a(X,S),
       not exceeded(X,S).
```

To place the limit on the size of integers satisfying property w we say

```
#reify(w).
:- max(X,w),X > 100.
```

Similarly for v. The method lacks the power of the DLT templates but it may be still interesting to see how far one can go in the reuse of rules with this simple mechanism. Another paper relevant to our work is (Gustafsson & Kvarnstrom 2004), in which the authors investigate the applicability of the object-oriented paradigm to modeling complex dynamic domains. This work contains ideas which are in many respect similar to that of $\mathcal{M}$. They have a hierarchy of classes; their *methods* are somewhat similar to our *axioms*; there is an intuitive correspondence between our *instances* and their *Call* macro, etc. Of course, unlike $\mathcal{M}$, their system, TAL-C, is based on Temporal Action Logic which causes some differences in the approaches. We plan to further study this work and see if some of its features and methodological insights can be incorporated in our work. In particular

[3] I borrowed the term from C. Baral who seems to use it in a substantially more general way.

we are interested in the way they call methods over intervals of time. Two papers, (Anwar, Baral, & Dzifcak 2006) and (Lifschitz & Ren 2006), presented at this conference are also closely related to our research. How much of this work can be used in addition to (instead of) the features of $\mathcal{M}$ presented here is the subject of future research. In the author's opinion much more experience is needed before the community will be able to come up with a definite solution to the problem of structuring even comparatively simple knowledge bases.

Acknowledgment

The author would like to thank Chitta Baral and Vladimir Lifschitz for useful discussions on the subject of this paper, and to one of the anonymous reviewers for drawing our attention to the paper (Gustafsson & Kvarnstrom 2004). He is also grateful to Nasa (contract NASA-NNG05GP48G) and Arda for supporting this research.

References

Anwar, S.; Baral, C.; and Dzifcak, J. 2006. Macros, macro calls and use of ensembles in modular answer set programming. AAAI 2006 Spring Symposium Series. (to appear).

Balduccini, M., and Gelfond, M. 2003a. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)* 3(4–5):425–461.

Balduccini, M., and Gelfond, M. 2003b. Logic Programs with Consistency-Restoring Rules. In Doherty, P.; McCarthy, J.; and Williams, M.-A., eds., *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, 9–18.

Baral, C., and Gelfond, M. 2005. Reasoning about Intended Actions. In *Proceedings of AAAI05*, 689–694.

Baral, C.; Gelfond, M.; and Scherl, R. 2004. Using answer set programming to answer complex queries. In *Proceedings of Workshop on Pragmatics of Question Answering at HLT-NAAC2004 (Human Language Technology - Annual Meeting for North American Association for Computational Linguistics)*.

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.

Calimeri, F.; Dell'Armi, T.; Eiter, T.; Faber, W.; Gottlob, G.; Ianni, G.; Ielpa, G.; Koch, C.; Leone, N.; Perri, S.; Pfeifer, G.; and Polleres, A. 2002. The DLV System. In Flesca, S., and Ianni, G., eds., *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*.

Calimeri, F.; Ianni, G.; Ielpa, G.; Pietramala, A.; and Santoro, M. 2004. A system with template answer set programs. In *Jelia06*, 693–697.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 365–385.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3(16):193–210.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence* 153:105–140.

Gustafsson, J., and Kvarnstrom, J. 2004. Elaboration tolerance through object-orientation. *Artificial Intelligence* 153:239–285.

Hayes, P. J., and McCarthy, J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. Edinburgh University Press. 463–502.

Kolvekal, L. 2004. Developing an Inference Engine for CR-Prolog with Preferences. Master's thesis, Texas Tech University.

Lifschitz, V., and Ren, W. 2006. Toward a modular action description language. AAAI 2006 Spring Symposium Series. to appear.

Lifschitz, V. 2000. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference*. 85–96.

Mueller, E. 2004. Understanding script-based stories using commonsense reasoning. *Cognitive Systems Research* 5(4):307–340.

Niemela, I., and Simons, P. 2000. *Extending the Smodels System with Cardinality and Weight Constraints*. Logic-Based Artificial Intelligence. Kluwer Academic Publishers. 491–521.

Shanahan, M. 1995. A circumscriptive calculus of events. *Artificial Intelligence* 77:249–284.