491 Knowledge Representation

Tutorial Exercise

The Action Language $\mathcal{C}$+

Question 1 *(based on a fragment from 2007 exam)*

How are the following expressed as static and/or fluent dynamic laws of the action language $\mathcal{C}$+? (Write the laws in the general form, without using any $\mathcal{C}$+ abbreviations such as 'inertial' and 'causes'.) In each case, what is the translation to (time-stamped) causal rules and logic program? (The original exam question did not ask for the logic programs.)

1. The default value of fluent f is v.
2. The fluent g is inertial. (g is not necessarily Boolean.)
3. There is no state in which a particular (spring-loaded) door is both open and closed (though it is always one or the other).
4. The action of pushing the door causes it to become open if is closed; pushing the door is not possible (executable) if the door is open.
5. If the door is closed, it remains closed by default ('inertia'); if it is open, it will be closed in the next state, by default.

Question 2 *(Yale Shooting Problem, with targets)*

Consider the following variant of the 'Yale Shooting Problem' (YSP). There are actions of loading a gun, aiming the gun at a person x, who then becomes the 'target', and shooting (i.e., pulling the trigger). Shooting a loaded gun when it is aimed at x results in x being dead (not alive). Shooting the gun also unloads it. The gun can be shot when it is not loaded but that has no effect, whether the target is alive or not.

Formulate this version of the YSP as an action description in the language $\mathcal{C}$+.

Use a multivalued fluent *target*=x. You should allow for the possibility that there is no target (e.g., by including *none* in the domain of *target*). For the action of aiming, you can either use Boolean action constants *aim(x)* or a single multi-valued action constant *aim* whose domain is the same as that of *target*. (It doesn't make much difference.)

Assume for simplicity that targets do not move. Aiming the gun at x means pointing it at x. And loading it means ensuring it is loaded: one can 'load' a gun that is already loaded.

Further: a gun cannot be loaded and aimed at the same time; a gun cannot be loaded and shot at the same time; a gun cannot be shot and aimed at the same time.

Assuming the action description has been translated to a logic program, how would you formulate a query to determine whether the sequence of actions *aim at a*, *load*, *aim at b*, *shoot* results in fluent *alive(b)* being false when in the initial state the gun is not loaded (and both a and b are alive)?

Suppose the (anonymous) shooter is happy if (and only if) everyone is dead.

How would you get a 'plan' for the goal where the shooter is happy?

Use a Boolean statically determined fluent *happy*.

Question 3 *(Yale Shooting Problem, with lights)*

Suppose there is also a light (represented by the Boolean fluent *on*).

- Shooting a gun when it is loaded kills the target, but only if the light is on. If the light is off, it is possible to shoot the gun but it will have no effect on the target, loaded or not.
- It is possible to load the gun while the light is off but not possible to aim the gun while the light is off.
- Whenever the light is off, there is no target (*target*=*none*).

Consider two variations: there is a Boolean action *toggle*: it switches the light to on if it is off, and to off if it is on.

Another possibility: the status of the light is exogenous.

Question 4 *(Yale Shooting Problem, trigger-happy)*

Suppose the person with the gun is trigger-happy: whenever the gun is loaded, this person shoots, whether there is currently a target or not.

Question 5

Make up your own variations. For instance ...

Yale Shooting Problem, with moving targets. Suppose that targets can move. There are (Boolean) action constants *move*(x). They represent that x moves (it does not matter where).

What happens if the target moves? What happens if the target moves as the gun is being shot?

491 Knowledge Representation

Tutorial Exercise

The Action Language $\mathcal{C}+$

Question 1

(1) default $f = v$ is shorthand for

caused $f = v$ if $f = v$

which translates to (for paths of length m):

$$f[i] = v \;\Leftarrow\; f[i] = v \qquad (0 \le i \le m)$$

In `clingo` syntax, and depending on the chosen representation for atoms, the logic program rule would be:

```
val(f,T,v) :- not -val(f,T,v), T = 0..maxT.
```

Whether definitions of `-val(f,T,v)` are required in addition depends on whether f is Boolean or multivalued and what its possible values ($dom(f)$) are. They are not specified in the question. (The original exam question did not ask for the logic program.)

(2) Suppose $dom(g) = \{v_1, \ldots, v_n\}$. The Boolean case is easy.
inertial g is shorthand for

caused $g = v_1$ if $g = v_1$ after $g = v_1$
$\vdots$
caused $g = v_n$ if $g = v_n$ after $g = v_n$

which translates to:

$$g[i{+}1] = v_1 \;\Leftarrow\; g[i{+}1] = v_1 \wedge g[i] = v_1 \qquad (0 \le i < m)$$
$$\vdots$$
$$g[i{+}1] = v_n \;\Leftarrow\; g[i{+}1] = v_n \wedge g[i] = v_n \qquad (0 \le i < m)$$

In `clingo` syntax, the logic program would include rules:

```
val(g,T+1,v1) :- not -val(g,T+1,v1), not -val(g,T,v1), T = 0..maxT-1.
    :
val(g,T+1,vn) :- not -val(g,T+1,vn), not -val(g,T,vn), T = 0..maxT-1.
```

or equivalently:

```
val(g,T+1,v1) :- not -val(g,T+1,v1), val(g,T,v1), T = 0..maxT-1.
    :
val(g,T+1,vn) :- not -val(g,T+1,vn), val(g,T,vn), T = 0..maxT-1.
```

We would also need $n(n-1)$ rules to define the `-val(g,T,v1)`, $\ldots$`-val(g,T,vn)` literals.

```
-val(g,T,v1) :- val(g,T,v2), T = 0..maxT.
      :
-val(g,T,v1) :- val(g,T,vn), T = 0..maxT.
                  :
                  :
-val(g,T,vn) :- val(g,T,v1), T = 0..maxT.
      :
-val(g,T,vn) :- val(g,T,vx), T = 0..maxT.  % vx denotes n-1'th value in dom(g)
```

(The original exam question did not ask for the logic program. No exam question would ask for logic programs with non-Boolean constants simply because of the amount of writing it would require.)

(3) There is no state in which a particular (spring-loaded) door is both open and closed (though it is always one or the other).

One way: let the (multi-valued) fluent *door* have domain $\{open, closed\}$.

Another way (it comes to the same thing): use a Boolean fluent *open*, and let 'closed' be represented by $\neg open$. (Or the other way round, obviously.) A fluent such as *open(door)* would also be OK.

With two Boolean fluents *open* and *closed* we would need to add the following additional constraints explicitly:

caused $\bot$ if $open \wedge closed$
caused $\bot$ if $\neg open \wedge \neg closed$

The logic program, `clingo` syntax, for third possibility (two separate Boolean fluents):

```
:- open(T), closed(T), T=0..Tmax.     % not both
:- -open(T), -closed(T), T=0..Tmax.   % but one or the other
```

Parts (4) and (5) are discussed in the lecture notes on translation from $\mathcal{C}+$ to logic program ('Addendum').

Question 2 In what follows x ranges over agent names $a, b, c, \ldots$.

Fluent constants: $alive(x)$, $loaded$ (Boolean), $target$ (domain: $a, b, c, \ldots, none$).

Action constants: *load, shoot, aim(x)* (all Boolean).

Instead of many Boolean constants *aim(x)* you might prefer to use a single constant *aim* with domain the same as *target*: $a, b, c, \ldots, none$.

> inertial $target$
> inertial $loaded$
> inertial $alive(x)$

Instead of inertial $alive(x)$ you might prefer to say that $alive(x)$ persists *by default* but $\neg alive(x)$ persists, and no default about it:

> caused $alive(x)$ if $alive(x)$ after $alive(x)$
> caused $\neg\, alive(x)$ after $\neg\, alive(x)$ *(once you are dead, you are dead)*

It doesn't make any difference in this example. (Because there is no action which brings a dead x back to life.)

And the rest:

> *load* causes *loaded*
> *aim(x)* causes $target{=}x$ (or $aim{=}x$ causes $target{=}x$)
> *shoot* causes $\neg\, alive(x)$ if $loaded \wedge target{=}x$
> *shoot* causes $\neg\, loaded$
> nonexecutable $load \wedge aim(x)$ (or nonexecutable $load \wedge aim{=}x$)
> nonexecutable $load \wedge shoot$
> nonexecutable $shoot \wedge aim(x)$ (or nonexecutable $shoot \wedge aim{=}x$)

You might perhaps prefer to write the *shoot* causes ... rule like this:

> *shoot* causes $\neg\, alive(x)$ if $loaded \wedge target{=}x \wedge \neg\, alive(x)$

Personally I don't like these references to 'causes' anyway. If you write it out as a fluent dynamic law in full you get

> caused $\neg\, alive(x)$ if $\top$ after $loaded \wedge target{=}x$

which makes sense whether or not you add the extra $\neg\, alive(x)$ condition to the body. (It makes no difference in this example.)

Note that: use of a single multi-valued action constant *aim* with the same domain as *target* automatically builds in the constraint that one can't aim at two different targets simultaneously. I chose to use (Boolean) action constants $aim(x)$ instead. You might think that we would therefore need to add some further constraints, viz:

> nonexecutable $aim(x) \wedge aim(y)$ if $x \neq y$

It would not be wrong to do this — indeed, I included them in my first attempt — but then I noticed that they are unnecessary. This is because $aim(x)$ causes $target{=}x$ and $aim(y)$ causes $target{=}y$; but *target* can't have two different values simultaneously, so there can't be a transition with $aim(x)$ and $aim(y)$ for any $x \neq y$.

For the query: the details obviously depend on the chosen `clingo` representation of atoms. First, formulate what is given:

```
problem_1 :-
  alive(a,0), alive(b,0), -loaded(0),
  val(aim,0,a), load(1), val(aim,2,b),
  shoot(3).
```

To determine whether this *necessarily* implies `-alive(b,4)` we try to satisfy the constraints:

```
:- not problem_1.
:- not -alive(b,4).
```

We set constant `maxT=4`. If there is no answer set ('unsatisfiable') then all paths of length 4 satisfying `problem_1` (assuming there are some) must have `-alive(b,4)`.

Because every answer set of the translated $\mathcal{C}+$ rules must give a (consistent and) complete valuation to all constants at all time stamps, the constraint could also be expressed:

```
:- not problem_1.
:- alive(b,4).
```

If there is no answer set ('unsatisfiable') then no path of length 4 satisfying `problem_1` can have `alive(b,4)`, and so (assuming there is one) must have `-alive(b,4)`.

If we try instead the constraints:

```
:- not problem_1.
:- -alive(b,4).
```

we generate all paths in which b is alive at time 4.

If we just use the constraint:

```
:- not problem_1.
```

we generate all paths satifying `problem_1` and so discover (amongst other things) whether b is alive at time 4 or not.

I tried the above in *iCCalc*. All paths of length 4 had `-alive(b,4)` (as you would expect) but there were many such paths, depending on how many potential targets there are. For instance, with $dom(target) = \{a, b, c, none\}$ there are two paths: one in which c is alive at time 0 and then throughout, and one in which c is not alive at time 0, and throughout. With another target d we get four paths, with another target e we get eight paths, and so on.

To get rid of all these combinations I defined a (statically determined) Boolean fluent:

default all_alive
caused $\neg\, all\_alive$ if $\neg alive(x)$

Defined this way, all_alive does not depend on how many potential targets there are: x ranges over a, b, ... The logic program looks like this:

```
 all_alive(T) :- not -all_alive(T), T=0..maxT.
-all_alive(T) :- target(X), -alive(X,T), T=0..maxT.
```

Now I looked for answer sets satisfying:

```
problem_2 :-
   all_alive(0), -loaded(0),
   val(aim,0,a), load(1), val(aim,2,b),
   shoot(3).
:- not problem_2.
```

iCCalc produced just one path, no matter how many potential targets were specified, with `-alive(b,4)` in each.

Planning example: *happy* is a Boolean statically determined fluent constant.

default $happy$
caused $\neg\, happy$ if $alive(x)$

As for all_alive, this definition does not depend on how many potential targets there are: x ranges over a, b, ... (*happy* is *not* equivalent to $\neg all\_alive$.)

The logic program:

```
 happy(T) :- not -happy(T), T=0..maxT.
-happy(T) :- target(X), alive(X,T), T=0..maxT.
```

For the 'plan', formulate what is wanted:

```
plan_1 :-
   loaded(0), happy(maxT).
:- not plan_1.
```

Pick some maximum length m for runs/paths/traces and then iteratively set constant $\texttt{maxT} = 0..m$ and look for answer sets (paths).

I tried this in *iCCalc*, with a guessed value of $m = 10$, and with $dom(target) = \{a, b, c, none\}$. *iCCalc* produced four plans of length 0. All of them had *happy* (everyone dead) in the initial state, and differed only in the value of *target* in the initial state. That is obvious in hindsight — those are clearly the shortest 'plans'.

So then I asked for answer sets satisfying

```
plan_2 :-
   all_alive(0), loaded(0), happy(maxT).
:- not plan_2.
```

iteratively, for $\texttt{maxT} = 0..10$. With $dom(target) = \{a, b, c, none\}$ as before this produced 24 solutions at time=8; again, many differed simply in the value of *target* in the initial state. Again, this was obvious in hindsight. Also obvious in hindsight is that none of these plans had $target = none$ in the initial state — $target = none$ in the initial state would force one more action to aim the gun, making the plan 9 steps long instead of 8 and by looking iteratively we find the shortest paths first.

So then finally I tried finding answer sets

```
plan_3 :-
   val(target,0,none), all_alive(0), loaded(0), happy(maxT).
:- not plan_3.
```

iteratively, for $\texttt{maxT} = 0..10$. Now, for targets a, b, c, there are 48 solutions, of length 9. Why so many solutions? Partly it is because the targets can be shot in any order: with three targets a, b, c to kill, there are 6 possible permutations. The other variation is whether, for each victim, one first loads the gun then aims, or first aims and then loads the gun. To see the effect suppose we add one further constraint:

caused $shoot$ if $loaded$

This is an action dynamic law which says that the gun is fired whenever (as soon as) *loaded* is true. One could also write instead (equivalently):

nonexecutable $\neg shoot$ if $loaded$

With this extra constraint, and targets a, b, c, there are indeed 6 solutions, all of length 9. With targets a, b, c, d, I got $24 = 4!$ solutions, of length 12. With targets a, b, c, d, e, I got $120 = 5!$ solutions, of length 15. And so on.

(Naturally I wouldn't expect you to do these calculations without access to something like *iCCalc* or `clingo`. Many of these points — for instance how to formulate the initial state — only become obvious when we try it and see what we get.)

Question 3 x ranges over agent names $a, b, c, \ldots$.

The signature is as in Question 2, but we add a fluent constant *light* (domain: *on off*). (Or, if you prefer, use a Boolean fluent *on*. It doesn't make any difference.)

$\mathsf{inertial}\ target$
$\mathsf{inertial}\ loaded$
$\mathsf{inertial}\ alive(x)$
$\mathsf{inertial}\ light$

$load\ \mathsf{causes}\ loaded$
$aim(x)\ \mathsf{causes}\ target = x$
$shoot\ \mathsf{causes}\ \neg\ alive(x)\ \mathsf{if}\ loaded \wedge\ target = x \wedge\ light = on$ % modified
% we don't need that shooting with light off has no effect

$\mathsf{nonexecutable}\ load \wedge\ aim(x)$
$\mathsf{nonexecutable}\ load \wedge\ shoot$

$\mathsf{nonexecutable}\ aim(x)\ \mathsf{if}\ light = off$
$\mathsf{caused}\ target = none\ \mathsf{if}\ light = off$ % optional but probably worth saying

Now for first version we add a (Boolean) action constant *toggle* and

$toggle\ \mathsf{causes}\ light = on\ \mathsf{if}\ light = off$
$toggle\ \mathsf{causes}\ light = off\ \mathsf{if}\ light = on$

For the second version, instead of the action *toggle* we make *light* 'exogenous': its value varies from state to state but we do not specify how — that is outside the system being modelled. exogenous *light* is shorthand for the pair of $\mathcal{C}+$ laws:

$\mathsf{caused}\ light = on\ \mathsf{if}\ light = on$
$\mathsf{caused}\ light = off\ \mathsf{if}\ light = off$

And we have to get rid of inertial *light* (otherwise it would never change from state to state.)

Question 4 ('*trigger-happy*')

This was already done in the discussion of Question 2. Add either the action dynamic law

$\mathsf{caused}\ shoot\ \mathsf{if}\ loaded$

or equivalently (in this example) the fluent dynamic law

$\mathsf{nonexecutable}\ shoot\ \mathsf{if}\ \neg loaded$ ($shoot\ \mathsf{causes}\ \perp\ \mathsf{if}\ \neg loaded$)

Question 5 ('*moving targets*')

There are many possible variations. What I had in mind was something like this:

$move(x)\ \mathsf{causes}\ target{=}none\ \mathsf{if}\ target{=}x$

The above does not deal with the case where the current target x moves, and in moving exposes some other target y standing behind. Not does it deal with the possibility that y moves into the line of fire.

So perhaps, instead of the above:

$move(x)\ \mathsf{may\ cause}\ target = y$ (for all $y \in dom(target)$, including $y = none$)

that is, in full:

$target = y\ \mathsf{if}\ target = y\ \mathsf{after}\ move(x)$

Similarly, the effects of shooting have to be adjusted. We could try, for example

$shoot \wedge \neg move(x)\ \mathsf{causes}\ \neg\ alive(x)\ \mathsf{if}\ loaded \wedge target{=}x$

that is, in full:

$\mathsf{caused}\ \neg\ alive(x)\ \mathsf{if}\ \top\ \mathsf{after}\ shoot \wedge \neg move(x) \wedge loaded \wedge target{=}x$

But again that does not deal with the possibility that someone else moves into the line of fire, or is exposed when x moves.

So perhaps better (y ranges over a, b, ...):

$\mathsf{caused}\ no\text{-}one\text{-}moves\ \mathsf{if}\ no\text{-}one\text{-}moves$
$\mathsf{caused}\ \neg no\text{-}one\text{-}moves\ \mathsf{if}\ move(x)$

$shoot \wedge no\text{-}one\text{-}moves\ \mathsf{causes}\ \neg alive(x)\ \mathsf{if}\ loaded \wedge target = x$
$shoot \wedge move(y)\ \mathsf{may\ cause}\ \neg alive(x)\ \mathsf{if}\ loaded \wedge alive(x)$

that is, in full:

$\mathsf{caused}\ \neg alive(x)\ \mathsf{if}\ \top\ \mathsf{after}\ shoot \wedge no\text{-}one\text{-}moves \wedge loaded \wedge target = x$
$\mathsf{caused}\ \neg alive(x)\ \mathsf{if}\ \neg alive(x)\ \mathsf{after}\ shoot \wedge move(y) \wedge loaded \wedge alive(x)$

You can, if you prefer, add another condition $\neg alive(x)$ to the first of the laws above, and in the second write $\neg no\text{-}one\text{-}moves$ in place of $move(y)$.

Of course there are many other possibilities.