Computational Intelligence

A Logical Approach

Problems for Chapter 3

Here are some problems to help you understand the material in Computational Intelligence: *A Logical Approach*. They are designed to help students understand the material and practice for exams.

This file is available in html, or in pdf format, either without solutions or with solutions. (The pdf can be read using the free acrobat reader or with recent versions of Ghostscript).

1 Defining a Simple Relation

Define the predicate *happy*(*P*, *D*) that is true when person *P* is happy on day *D*. A person is happy on a day if

- the person is a student and the day is a holiday, or
- the person is teaching a course that has a midterm on that day, or
- the person is David and the day is either Tuesday or Sunday.

You may use whatever constant symbols (e.g., "*david*") or predicate symbols (e.g., "*teaching*") you require. If the intended interpretation of a symbol isn't obvious you must give its intended interpretation.

Solution to the happy question

The simplest solution is:

```
happy(P,D) <- student(P) & holiday(D).
happy(P,D) <- teaching(P,C) & midterm(C,D).
happy(david,tuesday).
happy(david,sunday).
```

Alternatively, the last two clauses could be replaced by:

```
happy(david,D) <- tuesday(D).
happy(david,D) <- sunday(D).
```

This is probably better as there may be many different Tuesdays that we would want to consider as a Tuesday. We probaly don't want to do the same thing for the constant *david*, as we mean a particular David, not just anyone called David.

2 Adding to the Electrical Domain

Suppose we want to be able to reason about electric kettles plugged into the power outlets. Suppose the kettles need to be plugged in to a working power outlet, they need to be turned on, and be filled with water, in order to be heating.

Using CILog write axioms that let the system determine whether kettles are heating. Your program needs to be able to reason about multiple kettles. You should assume that the axioms are to be added to the axioms for the electrical domain.

You need to hand in

- a description of the intended interpretation of all symbols used.
- the CILog program that works. Your program should contain enough facts about specific kettles to test your axiomatization.
- a trace of your cilog program. Your trace should include enough information to verify your axiomatization is correct.

CILog code for the electrical environment is available as elect.pl.

Solution to Overhead Projector Problem.

This code is available as ohp.pl.

```
% heating(K) is true if kettle K is heating
heating(K) <-
   plugged_into(K,P) &
   live(P) &
   turned_on(K) &
   filled_with_water(K).

% kettle(K) is true if K is a kettle
kettle(k1).
kettle(k2).
kettle(k3).

% plugged_into(K,P) is true if K is plugged into power outlet P
plugged_into(k1,p1).
plugged_into(k2,p2).

% turned_on(K) is true if K is turned on
turned_on(k1).
turned_on(k3).

% filled_with_water(K) is true if K is filled with water
filled_with_water(k1).
filled_with_water(k2).
filled_with_water(k2).
```

Here is a trace of this program:

```
CILOG Version 0.12. Copyright 1998, David Poole.
CILOG comes with absolutely no warranty.
All inputs end with a period. Type "help." for help.
cilog: load 'elect.pl'.
CILOG theory elect.pl loaded.
cilog: load 'ohp.pl'.
CILOG theory ohp.pl loaded.
cilog: ask ohp_lit(OHP).
Answer: ohp_lit(ohp1).
  [ok,more,how,help]: more.
Answer: ohp_lit(ohp2).
  [ok,more,how,help]: more.
No more answers.
cilog: ask could_see_transparency(OHP).
Answer: could_see_transparency(ohp1).
  [ok,more,how,help]: more.
No more answers.
cilog: ask can_see(Tran).
Answer: can_see(lect2_3).
  [ok,more,how,help]: more.
No more answers.
cilog:
```

3 House Plumbing

Consider the domain of house plumbing represented in the diagram of Figure 1.

In this example constants $p1$, $p2$ and $p3$ denote cold water pipes. Constants $t1$, $t2$ and $t3$ denote taps and $d1$, $d2$ and $d3$ denote drainage pipes. The constants *shower* denotes a shower, *bath* denotes a bath, *sink* denotes a sink and *floor* denotes the floor. Figure 1 is intended to give the denotation for the symbols.

Suppose we have as predicate symbols:

- *pressurised*, where *pressurised*(P) is true if pipe P has mains pressure in it.
- *on*, where *on*(T) is true if tap T is on.
- *off*, where *off*(T) is true if tap T is off.
- *wet*, where *wet*(B) is true if B is wet.
- *flow*, where *flow*(P) is true if water is flowing through P.
- *plugged*, where *plugged*(S) is true if S is either a sink or a bath and has the plug in.
- *unplugged*, where *unplugged*(S) is true if S is either a sink or a bath and has the plug in.

The file plumbing.pl contains a CILog axiomatization for how water can flow down drain $d1$ if taps $t1$ and $t2$ are on and the bath is unplugged.

(a) Finish the axiomatization for the sink in the same manner as the axiomatization for the bath. Test it in CILog.

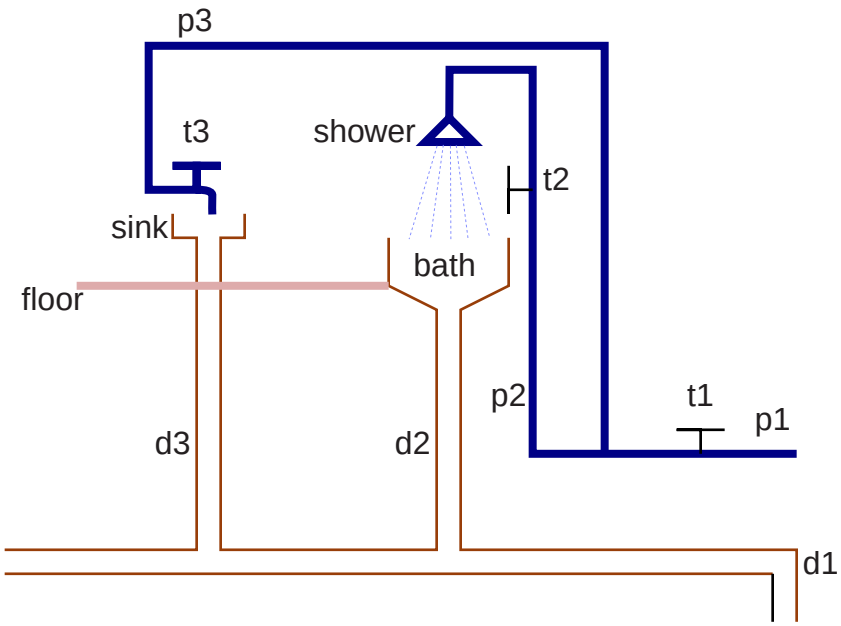

Figure 1: The Plumbing Domain

(b) Axiomatize how the floor is wet if the sink overflows or the bath overflows. They overflow if the plug is in and water is flowing in. You may invent new predicates as long as you give their intended interpretation. [Assume that the taps and plugs have been in the same positions for one hour; you don't need to axiomatize the dynamics of the turning on taps and inserting and removing plugs.] Test it in CILog.

(c) Suppose there is a hot water system is installed to the left of tap $t1$. This has another tap in the pipe leading into it, and supplies hot water to the shower and the sink (there are separate hot and cold water taps for each). Add this to your axiomatization. Give the denotation for all constants and predicate symbols you invent. Test it in CILog.

You need to hand in a complete listing of your program, including the intended interpretation for all symbols used and a trace of the CILog session to show it runs.

Solution to part (a).

Finish the axiomatization for the sink in the same manner as the axiomatization for the bath.

```
pressurised(p3) <-  on(t1) & pressurised(p1).
wet(sink) <- on(t3) & pressurised(p3).
flow(d3) <- wet(sink) & unplugged(sink).
flow(d1) <- flow(d3).
```

Solution to part (b).

Axiomatize how the floor is wet if the sink overflows or the bath overflows. They overflow if the plug is in and water is flowing in. You may invent new predicates as long as you give their intended

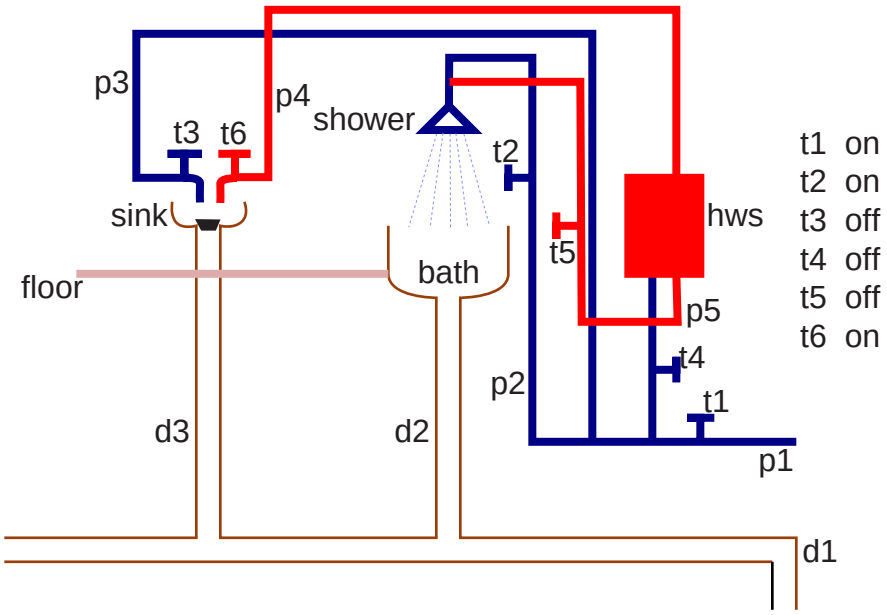

Figure 2: The Plumbing Domain with hot water

interpretation.

Here is the minimal set of clauses:

```
wet(floor) <- wet(sink) & plugged(sink).
wet(floor) <- wet(bath) & plugged(bath).
```

Solution to part (c).

Suppose there is a hot water system is installed to the left of tap $t1$. This has another tap in the pipe leading into it, and supplies hot water to the shower and the sink (there are separate hot and cold water taps for each). Add this to your axiomatization. Give the denotation for all constants and predicate symbols you invent.

The denotation is given in the diagram of Figure 2.

Here is the simplest axiomatization:

```
pressurised(hws) <- on(t4) & pressurised(p2).
pressurised(p4) <- pressurised(hws).
pressurised(p5) <- pressurised(hws).
flow(shower) <- on(t5) & pressurised(p5).
wet(sink) <- on(t6) & pressurised(p4).
```

The file plumbing2.pl contains a full axiomatization.

4 Designing Video Presentations

In this question you are to write a CILog knowledge base for the design of custom video presentations.

You should assume that the video is annotated using the relation

```
segment(SegId, Duration, Covers)
```

where *SegId* is an identifier for the segment. (In a real application this will be enough information to extract the segment from the video disk). *Duration* is the time of the segment (in seconds). *Covers* is a list of topics that is covered by the video segment. An example of a video annotation is the database:

```
segment(seg0,10,[welcome]).
segment(seg1,30,[skiing,views]).
segment(seg2,50,[welcome,computational_intelligence,robots]).
segment(seg3,40,[graphics,dragons]).
segment(seg4,50,[skiing,robots]).
```

A presentation is a sequence of segments. You will represent a presentation by a list of segment identifiers.

(a) Axiomatize a predicate

```
presentation(MustCover, Maxtime, Segments).
```

That is true if *Segments* is a presentation whose total running time is less than or equal to *Maxtime* seconds, such that all of the topics in the list *MustCover* are covered by a segment in the presentation. The aim of this predicate is to design presentations that cover a certain number of topics within a time limit.

For example, given the query:

```
cilog: ask presentation([welcome,skiing,robots], 90, Segs).
```

should at least return the two answers (perhaps with the segments in the other order):

```
Answer: presentation([welcome, skiing, robots], 90, [seg0, seg4]).
Answer: presentation([welcome, skiing, robots], 90, [seg2, seg1]).
```

Two procedures you may find useful are:

```
% member(E,L) is true if E is in list L
member(A,[A|R]).
member(A,[H|L]) <-
   member(A,L).

% notin(E,L) is true if E is not in list L
notin(E,[]).
notin(A,[B|L]) <-
   A \= B &
   notin(A,L).
```

(b) What is required for part (a) is reasonably straightforward. However, this example domain will be used for future problems, so it is worthwhile thinking about what you may want in such a presentation design program.

Assuming you have a good user interface and a way to actually view the presentations, list *three* things that the above program doesn't do that you may want in such a presentation system.

[There is no right answer for this part, you need to be creative to get full marks].

Solution to part (a).

I have four different solutions:

1. For the first solution, a segment is chosen that covers the first topic, and all of the topics that this segment also covers are removed from the list of topics left to cover.

2. The second solution is a directly recursive definition with no predicates defined other than in the question.

3. The third solution uses an iterative method, where we define another predicate *add_to_presentation* that adds to existing an presentation to also include more topics. Presentation then asks to add the topics it must cover to an empty presentation.

4. The fourth solution generates presentations that satisfy the length criterion, and then tests them to see if they cover all of the topics. This is the only one that defines more than one extra predicate.

Solution to part (b).

Here are some ideas:

- Maybe we want some measure of quality rather than just a time limit (e.g., we may want the shortest presentation).

- We may want to return the best presentation first.

- We may want a sensible order to the presentation. (The current versions don't worry about the order of the clips within a presentation).

- We may want some continuity for the presentations; the video should flow, and be coherent.

- What presentation a person wants may not only depend on what they are interested in, but what they already know about.

- The terms used to describe the topics for the user may not necessarily match with the topics in the segments database. For example, they may just want something about CS research, in which case either computational intelligence or graphics would be OK. The system should do this automatically (the user needn't know that there is computational intelligence research going on).

- There should still be a presentation if some topic cannot be covered. In this case we want the best presentation.

- The user may not know what sorts of things are in the video, even though they may be interested in it. We would like to implement “give me an interesting video that matches what I am interested in”.