Logical Control of an Elevator with Defeasible Logic

Michael A. Covington

Senior Member, IEEE

Artificial Intelligence Center

The University of Georgia

Athens, GA 30602-7415 U.S.A.

(E-mail mc@uga.edu; fax 706 542-8864)[1]

1999 February 16

[1]The author would like to thank Donald Nute and David Billington for assistance and encouragement.

Abstract

The elevator control program described in this journal by Dyck and Caines [2] can be implemented more concisely in d-Prolog, a defeasible logic programming system developed by Nute [3, 4, 5]. In defeasible logic, more specific rules take precedence over more general ones. Thus, the d-Prolog programmer can state general rules and then give explicit exceptions, just as humans do when explaining complex regularities to each other.

Keywords: logic modeling, defeasible logic, defaults, elevator

1 Introduction

Human beings find it natural to explain complex situations by stating general rules followed by exceptions, each of which overrides the rules that it appears to contradict. For example, birds fly, but ostriches are birds, and ostriches don't fly. In classical logic, these three statements lead to a contradiction because as soon as you encounter an ostrich, you infer both that it flies and that it does not fly. In defeasible logic, however, the rule about ostriches overrides the rule about birds because it is more specific.

There are two reasons why humans use defeasible logic. First, human knowledge is imperfect. Classical logic specifies how to reason when all applicable facts are known with complete certainty. Defeasible reasoning is for reasoning when relevant facts may be unknown or uncertain — the usual human situation. Defeasible logic is logic that can change its mind, i.e., logic in which new information can override earlier conclusions [3, 4, 5].

Second, defeasible knowledge representations are often more concise. In classical logic, every rule must enumerate everything that could possibly affect its conclusion. Defeasible logic can say, "unless a more specific rule applies." As just noted, in defeasible logic you can say "Birds fly," and then add, "Ostriches are birds that don't fly," without creating a contradiction. Crucially, you don't have to go back and change the first rule to "Birds fly unless they are ostriches."

Like humans, automatic control systems often deal with situations that are conveniently described in terms of general rules with specific exceptions. The d-Prolog theorem prover [4] implements defeasible logic as a computer programming language; it is an extension of Prolog [5] and the full power of

Prolog is available. d-Prolog programs can be compiled into lookup tables for execution on low-end microcontrollers [1].

2 The elevator problem

Dyck and Caines [2] give a set of logical axioms for controlling an elevator. They assume that a separate routine, outside the theorem prover, maintains a queue of requests, each of which has a steadily increasing "frustration level" (age). Whenever the elevator reaches a floor at which it has been requested to stop, the doors open and the request is removed from the queue.

The job of the theorem prover is to deduce whether the elevator should stand still, move up, or move down, according to the following rules:

(1) By default, stand still.

(2) But if there are any requests in the queue, service the oldest request.

(3) But if there is a request for floor 4, service it immediately because floor 4 is the emergency entrance of a hospital.[1]

(4) But if there is a fire, stand still (at a floor, with the doors open, of course).

In Dyck and Caines' implementation, rule (2) splits into four cases: the oldest request could come from inside or outside the elevator, and it could refer to a floor above or below the current one. Without changing the logical

[1] One might want to suppress stops at intermediate floors in this case, but neither their algorithm nor mine does so.

nature of the problem, I make two simplifications. My elevator simply takes requests to stop at particular floors, without caring whether they come from the control panel inside the elevator or the call buttons outside, and my theorem prover deduces what floor the elevator should move toward, leaving it to a separate routine to decide whether this entails going up or down.

3 Defeasibility

The defeasible structure of this set of rules is evident; each rule is an exception to those that precede it. Dyck and Caines, using classical logic, cannot express this defeasibility directly. Instead, each of their control axioms specifies the conditions under which it does *not* apply, as well as those under which it does. As a result, many conditions are stated in more than one place. Schematically, their renderings of the four rules look like this:

(1) $\neg\mathbf{Fire} \wedge \neg\mathbf{Emergency} \wedge \mathbf{Pf}(\mathbf{Max}) = 0 \rightarrow U = 0.$

(2) $\neg\mathbf{Fire} \wedge \neg\mathbf{Emergency} \wedge \neg\mathbf{Pf}(\mathbf{Max}) = 0 \wedge \mathbf{Pl}(\mathbf{Max}) < \mathbf{Pn} \rightarrow U = 1$
(and three other cases).

(3) A series of axioms dealing with the special status of floor 4.

(4) $\mathbf{Fire} \rightarrow U = 0.$

Separate axioms identify the oldest request (**Max**).

For the full set of axioms and an explanation of the notation, see [2]; the important thing here is that ¬**Fire** and ¬**Emergency** are necessary in all rules that do not pertain to fires or hospital emergencies. Also, rule (2)

restates all the conditions of rule (1), negating the one that is not already negated.

Using d-Prolog, and inferring the destination rather than the direction of travel, the rules are encoded much more simply:

(1) `stop := true.`

(2) `move(X) := oldest_request(X).`

(3) `move(4) := requested(4,_), oldest_request(_).`

(4) `stop :- fire.`

Here '`:=`' means 'if, defeasibly,' and '`:-`' means 'if, absolutely.' Translated into English, these rules say:

(1) By default, stop (stand still).

(2) But if `oldest_request(X)` is true for some `X` (i.e., there are requests in the queue), move toward floor `X`.

(3) But if `requested(4,_)` is true, move toward floor 4, regardless of the status of `oldest_request`. (Here '`_`' means 'ignore this argument' — the age of the request for floor 4 and the floor of the oldest request.)

(4) Regardless, if `fire` is true, stop.

If it is necessary to deduce the direction of travel, one can add two more axioms in plain Prolog:

```
move_up   :- move(D), current_floor(C), D>C.
move_down :- move(D), current_floor(C), D<C.
```

```
% Inputs to the logic engine

requested(3,1). % requested(Floor,AgeOfRequest)
requested(2,2).
requested(0,3).
neg fire.       % is there a fire?

% Definition of "oldest request", in conventional Prolog:

oldest_request(Floor) :-
  requested(Floor,Age),
  \+ (requested(AnotherFloor,AnotherAge), AnotherAge > Age).

% Rules (in d-Prolog):

stop := true.

move(X) := oldest_request(X).

move(4) := requested(4,_), oldest_request(_).

stop :- fire.

pairwise_incompatible([stop,move(0),move(1),move(2),
                                          move(3),move(4)]).

% Code to display the results:

demo :- @@ [stop,move(0),move(1),move(2),move(3),move(4)].
```

Figure 1: d-Prolog implementation of elevator control rules.

Figure 1 shows the complete program, which also contains the declaration

```
pairwise_incompatible([stop,move(0),
    move(1),move(2),move(3),move(4)]).
```

to tell the theorem prover that `stop`, `move(0)`, `move(1)`, etc., are incompatible conclusions, i.e., any line of reasoning that implies that one of them is true also implies that all of the others are false. Otherwise the theorem prover would be unaware that the rules potentially conflict.

The theorem prover resolves conflicts as follows. Absolute (‘`:-`’) rules always override defeasible (‘`:=`’) rules. Thus, whenever `fire` is true, the theorem prover will infer `stop` and will not infer anything that conflicts with `stop`.

When two defeasible rules conflict, the more specific one wins out. For example, when `oldest_request(2)` is true, the premises of both rule (1) and rule (2) are satisfied. Rule (2) wins out because its premise, `oldest_request(X)`, is more specific than the premise of rule (1), `true`. By “more specific” we mean “true in a proper subset of the situations.” Thus, rule (2) wins out because its premise takes more information into account.

Likewise, when, for example, `oldest_request(2)` and `requested(4,1)` is true — that is, the oldest request is for floor 2 but there is also a request for floor 4 with an age of 1 — then rules (1), (2), and (3) are all satisfied, but rule (3) wins out because it takes into account all the premises of rules (1) and (2) plus another, more specific, premise of its own.

4 Another approach

Rule (3) should say, "If there is a request for floor 4, go there." But if encoded as

```
move(4) := requested(4,_).
```

it will not override rule (2) because the theorem prover will not recognize it as more specific. That is why it was encoded as:

```
move(4) := requested(4,_), oldest_request(_).
```

specifically mentioning `oldest_request`. This encoding is not unnatural; expressing it in English, we might say, "regardless of `oldest_request`." It does, however, trigger an unnecessary computation to find the oldest request.

One alternative is to add a premise to rule (2) instead of rule (3), thus:

(2) `move(X) := oldest_request(X), neg requested(4,_).`

(3) `move(4) := requested(4,_).`

Now rule (2) cannot be satisfied when there is a request for floor 4, and the question of precedence does not arise. However, this encoding still lacks elegance because the premise of rule (3) is still redundantly encoded in rule (2).

A better alternative is to tell the inference engine explicitly which rule should win out, thus:

```
move(X) := oldest_request(X).
```

```
move(4) := requested(4,_).


sup( (move(4) := requested(4,_)),
     (move(X) := oldest_request(X)) ).
```

The `sup` declaration states explicitly that rule 3 is superior to rule 2. As d-Prolog is presently implemented, the `sup` declaration has to quote, entire, the rules to which it applies, but a more concise notation could easily be provided. Indeed, for programming embedded systems, one may want to have a mode in which sequence determines superiority, i.e., all later rules are superior to all the rules that precede them.

5 Implicit defeasibility of conventional computer programming

If sequence determines superiority, a defeasible logic program resembles a conventional series of nested **if–then–else** statements such as this:

if requested(4) **then**
  move(4)
else if oldest_request(X) **then**
  move(X)
else
  stop.

This is not classical logic because the conditions for the second **if** and the final **else** are not locally explicit; a purely classical set of rules would be:

requested(4) $\rightarrow$ move(4)

($\exists$ X) oldest_request(X) $\wedge$ $\neg$ requested(4) $\rightarrow$ move(X)

$\neg$[($\exists$ X) oldest_request(X)] $\wedge$ $\neg$ requested(4) $\rightarrow$ stop

where each rule explicitly mentions the conditions under which it could be overridden. In conventional computer programming, as in defeasible logic, the overriding of rules depends on context. The difference is that defeasible logic can use the internal structure of the rules, not just the place where they appear in the program, to determine which one has precedence.

References

[1] Michael A. Covington, "Defeasible Logic on an Embedded Microcontroller," *Proceedings, Tenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE)*, 1997.

[2] Derek N. Dyck and Peter E. Caines, "The logical control of an elevator," *IEEE Trans. Automat. Contr.*, vol. 40, pp. 480–486, 1995.

[3] Donald Nute Basic defeasible logic. In *Intensional logics for programming,* ed. L. Fariñas del Cerro and M. Penttonen, pp. 125–154. Oxford: Oxford University Press, 1992.

[4] Donald Nute. d-Prolog: an implementation of defeasible logic in Prolog. In *Non-monotonic extensions of logic programming: theory, implementation, and applications,* ed. J. Dix, L. M. Pereira, and T.

Przymusinski, pp. 161–182. Research report 17/96, Institut für Informatik, University of Koblenz-Landau, 1996.

[5] Donald Nute Defeasible Prolog. In M. Covington, D. Nute, and A. Vellino, *Prolog programming in depth,* 2nd ed., pp. 345–405. Upper Saddle River, N.J.: Prentice-Hall, 1997.