Logical Control of an Elevator with Defeasible Logic

Michael A. Covington

Abstract—**The elevator control program described in this journal by Dyck and Caines [4] can be implemented more concisely in d-Prolog, a defeasible logic programming system developed by Nute [5], [7], [8]. To demonstrate this, the program is recast, first into ordinary Prolog and then into d-Prolog.**

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.

Index Terms—**Defaults, defeasible logic, elevator, logic modeling.**

I. 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 they imply both that an ostrich 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. In defeasible logic, new information can override earlier conclusions [5], [7], [8].

Second, defeasible knowledge representations are often more concise. In classical logic, every rule must enumerate everything that could possibly affect its conclusion. In defeasible logic, every rule has the implicit provision, "unless a more specific rule applies." As just noted, in defeasible logic one can say "Birds fly," and then add, "Ostriches are birds that don't fly," without creating a contradiction. Crucially, it is not necessary 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 [7] implements defeasible logic as a computer programming language; it is an extension of Prolog [8] and the full power of Prolog is available. d-Prolog programs can be compiled into lookup tables for execution on low-end microcontrollers [2].

II. The Elevator Problem

Dyck and Caines [4] 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 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.

III. 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 the following.

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; the request can be for a higher or lower floor and can come from inside or outside the elevator).
3) A series of axioms dealing with the special status of floor 4.
4) $\mathbf{Fire} \rightarrow U = 0$.

Here **Max** is the oldest request pending; $\mathbf{Pf}(\mathbf{Max})$ is its age (frustration level), 0 if there are no requests in the queue; $\mathbf{Pl}(\mathbf{Max})$ or $\mathbf{Pd}(\mathbf{Max})$ is the floor requested by **Max**; and **Pn** is the current position of the elevator; U is the control signal, 0 to stand still, 1 to go down, and 2 to go up. **Fire** and **Emergency** are true if, respectively, there is a fire or a request to go to the hospital (floor 4). The symbols $\neg$, $\wedge$, and $\rightarrow$ mean, respectively, "not," "and," and "implies."

Simplifying the control signal so that it specifies the floor to go to, rather than the direction of motion, these rules can be expressed in standard Prolog [3] as:

```
1) stop :- \+ move(_).
2) move(X) :- oldest_request(X),
                \+ requested(4,_),
                \+ fire.
3) move(4) :- requested(4,_),
                \+ fire.
4) stop :- fire.
```

That is: 1) stop if no move is requested; 2) move toward floor X if it is the oldest request and there is no fire or request for floor 4 (the hospital); 3) move toward floor 4 if it has been requested, unless there is a fire; and 4) stop if there is a fire.

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

Manuscript received February 22, 1999; revised November 27, 1999. Recommended by Associate Editor, M. Polycarpou.

The author is with the Artificial Intelligence Center, The University of Georgia, Athens, GA 30602-7415 USA (e-mail: mc@uga.edu).

Publisher Item Identifier S 0018-9286(00)06075-X.

Here `stop` and `move(N)` (move toward floor N) are the control signals. Outside the program, we specify that if both `stop` and `move(X)` (for some X) are inferred, `stop` takes precedence.

The symbol `:-` means "if;" the comma means "and;" `\+` means "not" (more precisely, "cannot derive"); and `_` is a dummy variable that matches any value. Thus, for example, `\+ move(_)` is true if `move(X)` cannot be inferred for any value of X.

The predicate `oldest_request(X)` means "the oldest request in the queue is for floor X;" `requested(X,F)` means "there is a request for floor X with age (frustration level) F;" and `fire` means "there is a fire."

The crucial drawback of Dyck and Caines' axioms and of my Prolog rules is that each rule mentions not only the conditions that trigger it, but also, negatively, the conditions under which some other rule should apply instead. For example, axioms 1) and 2) contain $\neg$**Fire** and $\neg$**Emergency** to ensure that axioms 3) and 4) take precedence when there is a fire or emergency. If rules 3) and 4) would be absent, these parts of rules 1) and 2) would be unnecessary.

IV. Applying Defeasible Logic

In d-Prolog, more specific rules automatically override less specific rules. Thus, rules do not, in general, need to contain, negated, the conditions of other rules:

```
1) stop := true.
2) move(X) := oldest_request(X).
3) move(4) := requested(4,_),
     oldest_request(_).
4) stop :- fire.
```

Here ":=" means "if, defeasibly;" the rest of the notation is as in Prolog. 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.

Fig. 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.

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`. "More specific" means "true in a proper subset of the situations." Thus, rule 2) wins out beoause its premise takes more information into account.

```
% 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)].
```

Fig. 1. d-Prolog implementation of elevator control rules.

Likewise, when `oldest_request(2)` and `requested(4,1)` are 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.

V. Explicit Superiority

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 natural in English one 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,_).
```

(Here neg is the d-Prolog negation operator.) 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, in their entirety, the rules to which it applies, but a more concise notation could easily be provided. Indeed, for programming control 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.

VI. Background of Defeasible Logic

Defeasible logic is one of several kinds of *nonmonotonic logic* [1], i.e., systems in which the set of conclusions can shrink as well as grow when the set of premises is enlarged. The purpose of nonmonotonic logic is, in general, to represent uncertain knowledge. However, I am using it primarily to represent knowledge more concisely.

One of the best-known nonmonotonic logic systems is the *default logic* of Reiter [12], in which rules have the form

$$\frac{A : B}{C}.$$

This means, "If A is true and B could be assumed true without contradiction, then conclude C." In effect, default logic adds premises to the rules. Thus, conciseness is not one of its advantages.

Defeasible logic, by contrast, achieves nonmonotonicity by constraining the inference process and was specifically designed for reasonably efficient implementation [6], [11]. In defeasible logic, when there are two premises with contradictory conclusions, the premise that is more specific (or is specified as superior) wins out. This has two advantages. First, defeasible logic can choose between conflicting defaults, thus solving some problems for which default logic is inadequate [6, pp. 365–367]. Second, defeasible rules have the conciseness advantage described at length earlier in this paper.

For soundness and completeness results for Nute's system, the basis of d-Prolog, see [9] and [10].

The computational cost of defeasible reasoning is admittedly high, since in order to draw a conclusion, the inference engine must determine not only that the conclusion is reachable, but also that it is not defeated by a line of inference leading to the opposite conclusion. However, defeasible logic programs can be compiled automatically into conventional logic programs or even into machine language. Thus, defeasible rule sets can be viewed as the input to a program generator rather than as programs to be executed directly (see [2]).

Acknowledgment

The author would like to thank D. Nute and D. Billington for assistance and encouragement.

References

[1] G. Brewka, J. Dix, and K. Konolige, *Nonmonotonic Reasoning: An Overview*. Stanford, CA: CSLI.

[2] M. A. Covington, "Defeasible logic on an embedded microcontroller," in *Proc. Tenth Int. Conf. Industrial and Engineering Appl. Artificial Intelligence and Expert Systems (IEA-AIE)*, 1997.

[3] M. A. Covington, D. Nute, and A. Vellino, *Prolog Programming in Depth*, second ed. Upper Saddle River, NJ: Prentice-Hall, 1997.

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

[5] D. Nute, "Basic defeasible logic," in *Intensional Logics for Programming*, L. Fariñas del Cerro and M. Penttonen, Eds. Oxford: Oxford University Press, 1992, pp. 125–154.

[6] ——, "Defeasible logic," in *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Oxford: Clarendon, 1994, vol. 3, pp. 353–395.

[7] ——, "d-Prolog: An implementation of defeasible logic in prolog," in *Non-Monotonic Extensions of Logic Programming: Theory, Implementation, and Applications*, J. Dix, L. M. Pereira, and T. Przymusinski, Eds: Institut für Informatik, Univ. of Koblenz-Landau, 1996, pp. 161–182.

[8] ——, "Defeasible prolog," in *Prolog Programming in Depth*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1997, pp. 345–405.

[9] D. Nute and K. Erk, "Defeasible logic graphs—I. Theory," *Decision Support Syst.*, vol. 22, pp. 277–293, 1998.

[10] D. Nute, Z. Hunter, and C. Henderson, "Defeasible logic graphs—II. Implementation," *Decision Support Syst.*, vol. 22, pp. 295–306, 1998.

[11] J. L. Pollock, "How to reason defeasibly," *Artif. Intell.*, vol. 57, pp. 1–42, 1992.

[12] R. Reiter, "A logic for default reasoning," *Artif. Intell.*, vol. 13, pp. 81–132, 1980.

[13] M. A. Covington, "Defeasible logic on an embedded microcontroller," Appl. Intell., to be published.