Applied Intelligence 13, 259–264, 2000

Defeasible Logic on an Embedded Microcontroller

MICHAEL A. COVINGTON
Artificial Intelligence Center, The University of Georgia, Athens, GA 30602-7415, USA
mc@ai.uga.edu

Abstract. Defeasible logic is a system of reasoning in which rules have exceptions, and when rules conflict, the one that applies most specifically to the situation wins out. This paper reports a successful application of defeasible logic to the implementation of an embedded control system. The system was programmed in d-Prolog (a defeasible extension of Prolog), and the inferences were compiled into a truth table that was encoded on a low-end PIC microcontroller.

Advantages of defeasible logic include conciseness and correct handling of the passage of time. It is distinct from fuzzy logic and probabilistic logic, addressing a different set of problems.

Keywords: microcontroller, logic programming, defeasible logic, defaults, embedded systems

1. Defeasible Logic

1.1. A Sample Problem

Consider the following rules for controlling an air conditioner:

(1) Run the air conditioner when the temperature is over 78°.
(2) Do not run the air conditioner when the AC line voltage is low, unless the temperature is above 81° (not just 78°). (This reduces demand when the power company is heavily loaded.)
(3) Never run the air conditioner if it was turned off less than 4 minutes ago (to protect the compressor).

To a human observer, these seem like perfectly reasonable rules, but according to classical logic, they are contradictory. Suppose the temperature is 79° and the line voltage is low. Rule (1) says to run the air conditioner and rule (2) says not to. Likewise, rule (3) conflicts with rules (1) and (2) whenever the temperature is high but the air conditioner has just shut off.

The reason that we humans do not notice the contradiction, or at least do not object to it, is that these rules conform to a familiar pattern of human reasoning known as *Defeasible Logic* [1]—logic that can change its mind, logic in which rules can have exceptions. We find it intuitively reasonable for the more specific rules to override the more general ones.

Defeasible logic is what we use when reasoning with incomplete information, so there is always the possibility of a more specific rule being invoked as we learn more about the situation. Classical logic is a model of how we reason when we are sure we know all the relevant facts—which, in real life, is not often. Classical conclusions are guaranteed true, but defeasible conclusions are only apparently true, based on the available evidence. To put this another way, defeasible logic is *non-monotonic* [2]; further information can cause it to abandon a conclusion that it would have reached from fewer premises.

As stated, the rules are not quite a full set. They do not say what to do when the temperature is below 78°, nor what to do when the temperature seems to be above 81° but below 78° (an impossible situation). We can remedy this by adding two more rules:

(4) Do not run the air conditioner unless the other rules say to do so.
(5) If the temperature is over 81°, then it is necessarily also over 78°.

Rules (3) and (5) are ABSOLUTE rules, as indicated by the words "never" and "necessarily"—no other information can override them. Rules (1), (2), and (4) are DEFEASIBLE rules, which means that they can be overridden by absolute rules and by defeasible rules that are more specific.

Defeasible logic provides a natural, concise way for human beings to describe the conditions under which things should happen. The remainder of this paper deals with how to translate defeasible rules into a program for an embedded controller.

1.2. Defeasible Logic and Human Thinking

Classical logic describes the right way to reason when all relevant premises are known with complete certainty. Human beings often find themselves in situations where they can't be sure they know all the relevant facts. In such situations they use defeasible logic or something like it.

Consider for example the "Tweety triangle" (for diagram see [3]). Tweety is an ostrich; ostriches are birds; birds fly; but ostriches don't fly. In classical logic, these premises lead to a contradiction, because they entail both that Tweety flies (as a bird) and that he doesn't fly (being an ostrich). However, in realistic human situations it is reasonable to treat ostriches as an exception to the usual properties of birds. If all we know is that Tweety is a bird, we infer, defeasibly, that he flies; if we then find out that he is an ostrich, we retract the conclusion.

There are situations in which "I don't know" is the correct conclusion. One of them is the "Nixon diamond" (again, see [3] for diagram). Normally, Quakers are pacifists and Republicans are not pacifists. Nixon is a Quaker and a Republican. Is he a pacifist? The correct answer, reached by Nute's defeasible logic, is that with only this information, one cannot tell. Defeasible logic does not just replace conclusions with different conclusions; it blocks lines of reasoning that should not be followed.

Can defeasible logic do anything that classical logic cannot do? Yes and no. By definition, since its output is a binary truth table, defeasible logic does nothing that other logics cannot do. The difference is in how it does it. Defeasible rules correspond to a natural kind of human thinking and provide a concise representation for complex sets of conditions. The whole reason for having programming languages, after all, is to accommodate the thinking patterns of human beings; the computer itself would be content with ones and zeroes.

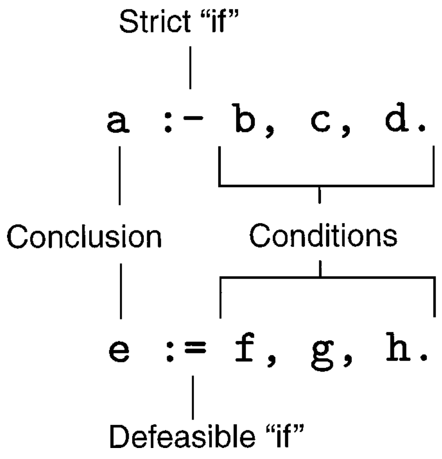

Figure 1. Strict and defeasible rules.

2. d-Prolog

2.1. Defeasible Inference Engine

A logic programming system that performs defeasible inference has been developed by Donald Nute [3, 4]. It is called d-Prolog and is implemented as an extension of Quintus Prolog. (Though related, d-Prolog is not the same as the defeasible system recently patented by Pollock [5, 6].)

As in Prolog, d-Prolog rules are of the form "A if B and C and D..." with any number of conjoined conditions (Fig. 1). Strict rules denote "if" by the symbol `:-` and cannot be overridden; if the conditions are true, so is the conclusion. Defeasible rules denote "if" by `:=` and are overridden by strict rules and by defeasible rules that take more information into account. For the complete mechanism see [3, 4]; not all of it is used in this paper.

Unlike Prolog, d-Prolog expresses negation explicitly by the prefixed operator `neg`. Thus, given a query X, the d-Prolog inference engine can conclude any of four different things:

- "Yes" (X can be inferred and `neg` X cannot).
- "No" (`neg` X can be inferred but X cannot).
- "I don't know" (neither X nor `neg` X can be inferred).
- "Contradiction" (both X and `neg` X can be inferred).

In addition, a "yes" or "no" answer can be absolute or defeasible depending on whether defeasible rules were used in deducing it.

Figure 2 shows how the air conditioner control rules are expressed in d-Prolog. The inputs are four bits,

```
run_ac := over_78.                            % Rule 1

neg run_ac := volt_low, over_78, neg over_81. % Rule 2

neg run_ac :- just_off.                       % Rule 3

neg run_ac := true.                           % Rule 4

over_78 :- over_81.                           % Rule 5

inputs([just_off,volt_low,over_81,over_78]).  % declarations needed by
outputs([run_ac]).                            % truth table generator
```

Figure 2. Air conditioner control rules in d-Prolog.

`just_off` (compressor was turned off less than 4 minutes ago), `volt_low` (line voltage is low), `over_81`, and `over_78` (temperature over 81° and 78° respectively). The conclusion is `run_ac` (run the air conditioner) or `neg run_ac` (don't run the air conditioner).

In Rule (4), `true` is a dummy condition that is always true. Thus, Rule 4 establishes a DEFAULT: "Do not run the air conditioner unless some other rule specifies otherwise." The d-Prolog inference engine knows that `true` takes no information into account and is therefore overruled by a defeasible rule with any other set of conditions.

Intermediate steps of reasoning are of course permitted, although there happen to be none in this example. A chain such as

```
a := b.
b := c.
c := d.
d := e.
```

(where `b`, `c`, and `d` are neither inputs nor outputs) is just as legitimate in d-Prolog as in ordinary Prolog.

3. Microcontroller Implementation

3.1. Hardware

Although this rule set is hardly large enough to demonstrate the full power of defeasible logic, a prototype air conditioner controller based on it has been implemented on a PIC16F84 microcontroller. The prototype controls an LED rather than an air conditioner, has switches rather than thermostats, and uses 4 seconds rather than 4 minutes as its timeout cycle. Nonetheless, it demonstrates that the control logic has been implemented correctly.

The PIC16F84 has 1K words of program memory, 64 bytes of RAM, and, in this application, runs at 0.1 MHz, and costs $6.48 in single quantities. It was chosen for convenience in programming and testing, since even cheaper processors are adequate for this application, such as the PIC12C508 ($1.50).

3.2. Truth Table

Because of its small size, the PIC does not run the d-Prolog inference engine directly. Instead, it stores a truth table that was generated by running d-Prolog on a Sun workstation. The truth table is generated as follows:

1. Take the list of inputs (`inputs` in Fig. 2) and negate some or all of its elements.
2. Assert the resulting set of premises (some affirmative and some negative).
3. Determine whether each of the outputs (also declared in Fig. 2) is true or false. If a contradiction or a "don't know" state is discovered, issue a warning message. (The need for rules (4) and (5) in the air conditioner system was in fact pointed out by the inference engine in this manner.)
4. Retract the temporarily asserted premises.
5. Backtrack to step 1 and try a different combination of negated and un-negated inputs.

The combinations of premises are tried in binary-number order (visible in Fig. 3), first 00000000 (all negated), then 00000001, 00000010, 00000011, and so on, ending with all premises non-negated. This makes it possible to use the input bit pattern as an offset into the table, adding it arithmetically to the starting address.

3.3. PIC Software

On the PIC, the truth table is implemented as a subroutine which, given an input bit pattern, returns an output bit pattern. In our situation there are 4 significant bits of input, `just_off`, `volt_low`, `over_81`, and `over_78`. These are stored in the four least significant bits of the input byte, and the truth table has $2^4 = 16$ entries. The output has only one significant bit, `run_ac`, which is stored in the lowest bit of the output byte.

The PIC has a particularly strict form of Harvard architecture, with separate memories for program and data. There is no way to read data from the program

```
; Generated by TRUTAB.PL
;   Input bits:  [0,0,0,0,just_off,volt_low,over_81,over_78]
;   Output bits: [0,0,0,0,0,0,0,run_ac]
;
; CPU: PIC16C84/F84
;
LOOKUP  ANDLW B'00001111'  ; eliminate unused bits
        ADDWF PCL,F        ; add offset to program counter
        RETLW B'00000000'  ; premises=[neg just_off,neg volt_low,neg over_81,neg over_78]
        RETLW B'00000001'  ; premises=[neg just_off,neg volt_low,neg over_81,over_78]
        RETLW B'00000001'  ; premises=[neg just_off,neg volt_low,over_81,neg over_78]
        RETLW B'00000001'  ; premises=[neg just_off,neg volt_low,over_81,over_78]
        RETLW B'00000000'  ; premises=[neg just_off,volt_low,neg over_81,neg over_78]
        RETLW B'00000000'  ; premises=[neg just_off,volt_low,neg over_81,over_78]
        RETLW B'00000001'  ; premises=[neg just_off,volt_low,over_81,neg over_78]
        RETLW B'00000001'  ; premises=[neg just_off,volt_low,over_81,over_78]
        RETLW B'00000000'  ; premises=[just_off,neg volt_low,neg over_81,neg over_78]
        RETLW B'00000000'  ; premises=[just_off,neg volt_low,neg over_81,over_78]
        RETLW B'00000000'  ; premises=[just_off,neg volt_low,over_81,neg over_78]
        RETLW B'00000000'  ; premises=[just_off,neg volt_low,over_81,over_78]
        RETLW B'00000000'  ; premises=[just_off,volt_low,neg over_81,neg over_78]
        RETLW B'00000000'  ; premises=[just_off,volt_low,neg over_81,over_78]
        RETLW B'00000000'  ; premises=[just_off,volt_low,over_81,neg over_78]
        RETLW B'00000000'  ; premises=[just_off,volt_low,over_81,over_78]
;
; End of generated code.
```

Figure 3. PIC microcontroller code generated by the d-Prolog program.

memory into data registers. Instead, data tables are implemented as successive RETLW instructions ("return with literal in W"), each of which contains a byte in the bottom 8 bits of the 14-bit instruction word. Accordingly, a table lookup subroutine adds the offset to the program counter, jumps to the appropriate RETLW, and returns with the appropriate table entry. Figure 3 shows how it is done; as a precaution, the irrelevant input bits are masked off to guarantee that execution does not jump outside the table.

The largest possible truth table, using this technique, has 8 input bits and up to 8 output bits, and occupies 256 bytes. If that's not enough, minor changes in implementation can accommodate much larger tables. If a truth table were all that was needed, it could reside in a programmable logic array rather than a microcontroller, but in typical applications, the services of a CPU are needed to gather input data and keep time.

4. Scaling to Larger Systems

The system described so far is so small that it would have been easy to implement with a conventional programming language or even assembly language. Clearly, if defeasible logic is advantageous, its advantages will show up in larger systems. Do they?

No large defeasible control systems have been built yet, but there are grounds for optimism. The knowledge systems in human minds are, arguably, defeasible, and they are large. Defeasibility is what makes it possible to learn exceptions without unlearning the general principles to which the exceptions apply.

As already mentioned, d-Prolog provides some simple validation by detecting cases in which the answer is either "don't know" or "contradiction." In this manner, the need for two of the rules in the sample system was discovered.

Other, deeper, kinds of validation and optimization are straightforward to implement. Given a defeasible rule, the d-Prolog system can identify all the other rules with which it interacts. (This is already done internally, and it would be straightforward to display the results.) Similarly, the system can trace the reasoning actuated by any particular set of inputs.

The Prolog-like notation of d-Prolog is not carved in stone. System builders may prefer to express rules "forward," as in

```
IF volt_low
   AND over_78
   AND NOT over_81
THEN
   NOT run_ac
```

rather than

```
neg run_ac :=
  volt_low, over_78, neg over_81.
```

Such a notation is easily implemented. In particular, the symbol := (intended to represent a double left arrow) may distract Pascal and Algol programmers; it can easily be changed to make the system more user-friendly.

Farther afield, a graphical representation for defeasible rules has been developed [3] and a system for entering rules into the computer graphically is being developed (Nute, personal communication).

Are there applications too complex for defeasible logic? Probably; there are applications too complex for anything, and in any case, d-Prolog does not exempt the system builder from the need for clean, modular design, though it provides more elbow room. What is important is that some applications come out substantially simpler in defeasible logic than in classical logic or conventional structured programming. These are the applications for which d-Prolog is suitable.

5. Further Issues

5.1. Persistence Through Time

Automatic control systems often face situations in which something *normally* remains unchanged, or changes in a particular way, as time passes, but normal behavior is not absolutely guaranteed.

This is exemplified by a classic problem of temporal reasoning known as the Yale Shooting Problem, which consists of the rules:

(1) Normally, a gun that is loaded at time t will still be loaded at time $t + 1$.
(2) Normally, a person who is alive at time t will still be alive at time $t + 1$.
(3) Normally, a person who is alive and is shot with a loaded gun at time t will be dead at time $t + 1$.

All three rules are defeasible: sometimes an adversary sneaks in and secretly unloads a gun, sometimes a person dies spontaneously, and some people survive gunshots. Ordinarily, though, we expect people to die when shot with a previously loaded gun, and to remain alive otherwise.

Now suppose the gun was loaded at t_0 and a person who is alive gets shot with that gun at t_0+1. Then is that person dead at $t_0 + 2$? Defeasible logic correctly infers "presumably, yes," because rule (3) is more specific than rule (2) and therefore overrides it.

Crucially, *defeasible logic is not just a logic of defaults*; *it is a logic that chooses intelligently between one default and another.* A logic based purely on defaults would be stymied in the same situation because it could not tell which of the three defaults should be overridden [7, 8].

The Yale Shooting Problem typifies a kind of reasoning that pervades embedded control. Things normally stay the same unless acted upon, and things that are acted upon normally change in the specified way.

Consider for example the task of feeding a sheet of paper into a printer. Instead of bullets in the gun, we need paper in the tray. Normally, (1) if n sheets are in the tray at time t, then n sheets are still available at $t+1$. But (2) if a sheet is printed at t, the number of sheets remaining at $t + 1$ is $n - 1$. And (3) if the tray has been opened, the number of sheets in it is unknown. Situation (2) overrides situation (1), and situation (3) blocks a conclusion without warranting a new conclusion; it is what Nute calls a *defeater*, somewhat like "A sick bird might not fly."

Embedded controllers perceive time in discrete units—indeed, one of the common ways to use a microcontroller is to let its watchdog timer reboot it several times a second, whereupon it wakes up, makes a decision, and goes back to sleep. Thus the sequence $t_0, t_0 + 1, t_0 + 2, \ldots,$ is exactly right for a microcontroller, even though time in the real world is a continuum.

5.2. Defeasible Versus Fuzzy Logic

A frequently asked question about defeasible logic is whether it is anything like fuzzy logic or probabilistic logic. The answer is, "Not really." Defeasible logic attacks a quite different set of problems and solves them in a different way.

There are two ways to use defeasible reasoning. We can use it simply as a more concise and human-friendly notation for formulas that could be expressed in classical logic. In that case, it contributes nothing to the power of an embedded system, but potentially a great deal to the ease of programming it. Or we can use defeasible logic to represent uncertain information, as in the Yale Shooting Problem. In the latter case, defeasible logic represents uncertainty in a quite different way than other technologies.

The purpose of fuzzy logic is to compute compromises numerically between conflicting conditions that are both true to a degree. This is not necessarily a matter of uncertainty; the premises may be perfectly certain but express judgments that are not binary. In practice, fuzzy logic programs are numerical models that are adjusted empirically, like other kinds of numerical models, to give the desired results [9, 10].

Probabilistic reasoning deals with premises that may or may not be true, but whose probabilities are known or estimated.

Neither of these is like defeasible reasoning. To say that a conclusion is defeasible is not to say that it is true only to a degree (as in fuzzy logic) nor that it has only a certain probability of being true. Defeasible logic makes no claims about likelihood; it only claims that if a conclusion is defeasible, further information can cause it to be overridden. To characterize the defeasibility further, one enumerates the kinds of premises that would cause the conclusion to be withdrawn. One need not know the likelihood of these premises actually turning out to be true.

Used in this way, defeasible logic models human reasoning from incomplete information. Instead of dealing precisely with every situation (as in classical logic), human beings make generalizations that cover most situations, then enumerate the exceptions by means of more specific rules. Defeasible logic allows the designer of an embedded control system to describe a complex truth table this way, using generalities and exceptions, rather than requiring exceptionless classical rules or numerical parameters.

6. Conclusions

Defeasible logic should take its place alongside fuzzy logic, probabilistic reasoning, and conventional computer programming in the embedded system designer's toolkit. It isn't the solution to every problem, but in the right cases, it provides a convenient design methodology that leads to rapid implementation. In particular, the literature on fuzzy logic sometimes expresses a wish for other non-classical reasoning techniques (see for example [11]), and defeasible logic is one of them.

Acknowledgments

An earlier version of this paper was presented at IEA-AIE 1997. I want to thank Donald Nute, David Billington, and Don Potter for encouragement and assistance with this project.

References

1. D. Nute, “Basic defeasible logic,” in *Intensional Logics for Programming*, edited by L. Fariñas del Cerro and M. Penttonen, Oxford University Press: Oxford, pp. 125–154, 1992.
2. G. Antoniou, *Nonmonotonic Reasoning*, MIT Press: Cambridge, Mass., 1997.
3. D. Nute, “Defeasible prolog,” in *Prolog Programming in Depth*, 2nd ed., edited by M. Covington, D. Nute, and A. Vellino, Prentice-Hall: Upper Saddle River, N.J., pp. 345–405, 1997.
4. D. Nute, “d-Prolog: an implementation of defeasible logic in Prolog,” in *Non-Monotonic Extensions of Logic Programming: Theory, Implementation, and Applications*, edited by J. Dix, L.M. Pereira, and T. Przymusinski, pp. 161–182. Research Report 17/96, Institut für Informatik, University of Koblenz-Landau, 1996.
5. J.L. Pollock, “How to reason defeasibly,” *Artificial Intelligence*, vol. 57, pp. 1–42, 1992.
6. J.L. Pollock, “Architecture for an artificial agent that reasons defeasibly,” United States Patent 5,706,406, January 6, 1998.
7. S. Hanks and D. McDermott, “Nonmonotonic logic and temporal projection,” *Artificial Intelligence*, vol. 33, pp. 379–412, 1987.
8. E. Sandewall and Y. Shoham, “Non-monotonic temporal reasoning,” in *Handbook of Logic in Artificial Intelligence and Logic Programming*, edited by D.M. Gabbay, C.J. Hogger, and J.A. Robinson, Clarendon Press: Oxford, vol. 4, pp. 439–498, 1995.
9. J. Bezdek, “Editorial: fuzzy models—what are they, and why?” in *Fuzzy Logic Technology and Applications*, edited by R.J. Marks II, IEEE: New York, pp. 3–7, 1992.
10. H. Surmann, A.P. Ungering, T. Kettner, and K. Goser, “What kind of hardware is necessary for a fuzzy rule based system?” in *Proceedings, Third IEEE Conference on Fuzzy Systems*, IEEE: New York, vol. 1, pp. 274–278, 1994.
11. R. Jager, H.B. Verbruggen, and P.M. Bruijn, “Demystification of fuzzy control,” in *Fuzzy Reasoning in Information, Decision and Control Systems*, edited by S.G. Tzafestas and A.N. Venetsanopoulos, Kluwer: Dordrecht, pp. 165–197, 1994.