Defeasible Prolog

Donald Nute
Artificial Intelligence Programs and Department of Philosophy
The University of Georgia, Athens, GA 30602, U.S.A
dnute@ai.uga.edu

d-Prolog is a nonmonotonic extension of the Prolog programming language. It is an implementation of *defeasible logic* (Nute 1991, 1992). I will describe the system and the utilities that support development of d-Prolog programs, making extensive use of simple examples to explain the logic of the system.

Many nonmonotonic formalisms use a special unary operator to mark peculiar conditions in rules. As long as all the conditions for the rule are satisfied, the consequent of the rule is detachable. For example, in autoepistemic logic we can always derive r from p, $\sim K \sim q$, and $p \wedge \sim K \sim q \supset r$. By contrast, d-Prolog and other defeasible systems (Loui 1987a, 1987b; Geffner 1992) use rules whose consequents may not be detachable even when their antecedents are derivable. Detachment of the consequent of one of these *defeasible* rules may be *defeated* by another rule. The competing rule may either *rebut* the first rule by supporting a conflicting consequent, or it may simply *undercut* the first rule by identifying a situation in which the rule does not apply (Pollock 1991). Defeasible logic uses strict rules, defeasible rules, and undercutting defeaters.

1 The basics

We begin by defining the language of d-Prolog. One unary functor `neg` and two binary infix functors `:=` and `:^` are added to Prolog. `neg` is a sound negation operator which we distinguish from the built-in negation-by-failure (NBF) operator `not`. Where `Atom` is an atomic clause, `Atom` and `neg Atom` are *complements* of each other. `neg Atom` can occur in either the Head or the Body of a rule. Clauses of the form `Head := Body` are called *defeasible* rules, and clauses of the form `Head :^ Body` are called *undercutting defeaters* or simply *defeaters*. A defeasible rule of the form `Head := true` is called a *presumption*. By contrast, ordinary Prolog rules are called *strict* rules.

Conclusions may be derivable either strictly or defeasibly. A conclusion is derivable strictly if and only if it is derivable from the facts and strict rules in the knowledge base alone. A clause `Goal` is strictly derivable, then, just in case the query `?- Goal.` succeeds. A conclusion is defeasibly derivable if it is derivable using all the clauses in the knowledge base including defeasible rules, presumptions, and defeaters. Thus, a conclusion is defeasibly derivable if it is strictly derivable. We introduce a new unary functor `@` to invoke the defeasible inference engine, and a clause `Goal` is defeasibly derivable just in case the query `?- @ Goal.` succeeds. We read `@ Goal` as 'Defeasibly, Goal' or as 'Apparently, Goal'.

Another unary functor `@@` is introduced to support exhaustive investigation of queries. We want a way to find out with a single query whether a *ground* atomic clause or its negation is either strictly or defeasibly derivable. In response to the query `?- @@ Goal`, d-Prolog will test for all these possibilities and give an appropriate report: 'definitely yes', 'definitely no', 'presumably yes', 'presumably no', or 'can't tell'.

When we have two strict rules with incompatible heads, arguably one of the rules is incorrect or the body of one or the other must be false. If the bodies of both are strictly derivable, we have no choice but to also infer the heads of both producing a contradiction. But when the bodies of both are only defeasibly derivable, we have a choice about what to do. We cannot tell from this situation which condition in which rule to reject, but we can at least

contain the damage by not inferring the consequent, even defeasibly, of either rule. Thus we make a design decision: a strict rule cannot be defeated if its body is strictly derivable; however, a strict rule is defeated by another strict rule with an incompatible head provided the body of the first rule is only defeasibly derivable and the body of the second is at least defeasibly derivable. When this happens, we say that the first rule is *rebutted* by the second since the second rule actually provides support for the incompatible conclusion. Notice that contradictions are impossible in Prolog since the only form of negation available is negation-by-failure. The same query can never both succeed and fail finitely. But as soon as we add sound negation, the possibility for contradictory knowledge bases arises. Since contradictions are localized even for that part of the system made up of facts and strict rules, d-Prolog is *paraconsistent*.

In contrast, defeasible rules may be rebutted by facts, strict rules, or other defeasible rules, and they may be undercut by defeaters. A fact that is incompatible with the head of a defeasible rule, or a rule (whether strict, defeasible, or only a defeater) whose head is incompatible with the head of a defeasible rule *competes* with that defeasible rule, and the defeasible rule and the competing fact or rule are *competitors*. The question whether a defeasible rule is rebutted or undercut only arises if the body of the rule is defeasibly derivable. Then the rule is only rebutted by a fact or rule to which it is not *superior*, and only undercut by a defeater to which it is not *superior*. Since facts and strict rules are superior to all defeasible rules, they rebut any defeasible competitors. We say a defeasible rule is *defeated* if it is either rebutted or undercut by some non-inferior competitor. A rule containing variables may be defeated for some instantiations of the variables but not for others.

As a first example, consider the following knowledge base.

```
born_in(X,usa) :- born_in(X,atlanta).
neg born_in(X,usa) :=
   native_speaker(X,greek).
born_in(stavros,atlanta) := true.
native_speaker(stavros,greek) := true.
```

For this knowledge base, d-Prolog responds to the query `?- @@ born_in(stavros,usa).` with the report 'presumably yes' because the strict rule is superior to the defeasible rule.

Another familiar example, the so-called Nixon Diamond, demonstrates how two defeasible rules may defeat each other. We represent this example in d-Prolog by the following facts and rules.

```
pacifist(X) := quaker(X).
neg pacifist(X) := republican(X).
quaker(nixon).
republican(nixon).
```

The correct response to the query

```
?- @@ pacifist(nixon)
```

is 'can't tell' since neither of the two defeasible rules is superior to the other and each is rebutted by the other. However, we could decide that in this kind of situation political party takes priority over religious affiliation. Then we can add the following clause to our d-Prolog knowledge base.

```
sup((pacifist(X) := quaker(X)),
  (neg pacifist(X) := republican(X))).
```

With this addition, the query

```
?- @@ pacifist(nixon).
```

produces the report 'presumably no'.

2 Specificity

A natural way to decide superiority of rules and to adjudicate conflicts between defeasible rules is to use *specificity*. Specificity is exemplified by the familiar Tweety Triangle:

```
flies(X) := bird(X).
neg flies(X) := penguin(X).
bird(X) := penguin(X).
penguin(tweety).
```

We can of course infer that Tweety is a bird. Does Tweety fly? We have conflicting rules, but we note that a penguin is a specific kind of bird. Since defeasible rules tell us what is typically or normally the case, and since we have evidence that Tweety is atypical at least so far as flying is concerned, we tentatively conclude that Tweety does not fly. However, penguins are birds, and a penguin that is a bird is in no way an atypical penguin. The

more specific rule is superior, and the query `?- @ neg flies(tweety).` succeeds.

In the Tweety Triangle, we can tell from the knowledge base that the rule for penguins is more specific than the rule for birds by the presence of the strict rule

```
bird(X) :- penguin(X).
```

As in the following example, specificity can also be signalled by a defeasible rule.

```
democrat(X) := southerner(X).
conservative(X) := southerner(X).
neg democrat(X) := conservative(X).
southerner(Nunn).
conservative(Nunn).
```

The query `?- @ democrat(Nunn).` succeeds because the rule for southerners is superior to the rule for conservatives, and this is so because southerners are a specific kind of conservative.

How does d-Prolog determine specificity? Basically, it looks at the bodies of the two conflicting rules to see if either can be derived from the other. In the case of Senator Nunn, d-Prolog tests whether the query `?- @ conservative(Nunn).` succeeds from a reduced knowledge base consisting of all the strict rules, all the defeasible rules (except presumptions), and all the defeaters in the knowledge base together with the clause `southerner(Nunn)`. Then it tries the converse. Since one test fails and the other succeeds, d-Prolog concludes that the rule for southerners is superior to the rule for conservatives. The reason for excluding facts and presumptions that do not occur in the bodies of the rules being compared should be clear. We know that Nunn is in fact a conservative. So if we used this fact, the query `?- @ conservative(Nunn).` would succeed from the body of any rule whatsoever.

The derivation of the body of one rule from the body of another proceeds in exactly the same way as the derivation of a conclusion from the complete d-Prolog knowledge base except that the set of facts and rules used in the derivation is different. This is accomplished by making the principle predicate in the inference engine, `def_der`, a binary predicate that takes a knowledge base as its first argument and a goal to be tested as its second argument. For top-level queries, the knowledge base is the complete d-Prolog knowledge base represented by the atom `root`. So the functor `@` is defined very simply:

```
@ Goal :- def_der(root,Goal).
```

The atom `root` is replaced by the body of a rule when specificity is being tested. Thus, in our Senator Nunn example, the actual queries involved in the test for specificity are

```
?- def_der(southerner(Nunn),
           conservative(Nunn)).
```

and

```
?- def_der(conservative(Nunn),
           southerner(Nunn)).
```

The first query succeeds and the second fails. The d-Prolog inference engine knows when it sees a knowledge base other than `root` to restrict itself to the items listed in the specified knowledge base together with the strict rules, defeasible rules except presumptions, and defeaters in the complete or 'root' knowledge base.

One more example will make it clearer how d-Prolog handles specificity. The first example concerns college students. We assume for our example that college students are normally adults and that normally adults are employed, but that college students normally are not employed. Furthermore, employed persons typically are self-supporting while college students typically are not self-supporting. Finally, we stipulate that Jane is a college student who is employed. Does Jane support herself? The rules and facts relevant to this example are:

```
adult(X) := college_student(X).
neg employed(X) :=
           college_student(X).
neg self_supporting(X) :=
           college_student(X).
employed(X) := adult(X).
self_supporting(X) := employed(X).
college_student(jane).
employed(jane).
```

We have two conflicting rules: one for college students and another for employed persons. Is either more specific than the other? Since college students are normally adults and adults are normally employed, it might appear at first glance that the rule for college students is more specific than the rule for employed persons. However, the query

```
?- def_der(college_student(X),
           employed(X)).
```

fails because the rule that college students normally are unemployed is more specific than the rule that adults normally are employed. So neither rule about self-support is more specific, each rule is rebutted by the other, and d-Prolog responds 'can't tell' to the query

```
?- @@ self_supporting(jane).
```

In d-Prolog, the use of specificity to determine superiority can be enabled or disabled. The query

```
?- spec.
```

toggles between enabling and disabling specificity. In response to the query, d-Prolog informs the user whether specificity has just been enabled or disabled. Even if specificity is enabled, the programmer can still add clauses for the predicate sup to force resolution of conflicts between rules where specificity fails to determine superiority. When d-Prolog is loaded, specificity is enabled by default.

3 Undercutting rules

We have considered several examples involving defeasible rules. In the English readings of these rules, we have used qualifying expressions like 'normally' or 'typically' to signal that the rules are defeasible. Each defeasible rule provides some evidence for its consequent or, in Prolog terminology, its head. Undercutting defeaters are quite different. An example of an undercutting defeater is

```
neg flies(X) :^ sick(X).
```

which we might read as 'A sick creature might not fly' or as 'The fact that a creature is sick undercuts other evidence we may have that it can fly'. Illness is not presented as a reason for concluding positively that a creature does not fly; it is only presented as a reason for rejecting what would otherwise be take as evidence that the creature flies.

An example is in order. A variation of the Tweety Triangle, this example concerns penguins that have been genetically altered to have large wings and correspondingly large flight muscles.

```
flies(X) := bird(X).
neg flies(X) := penguin(X).
flies(X) :^ altered_penguin(X).
bird(X) :- penguin(X).
penguin(X) :- altered_penguin(X).
altered_penguin(chirpy).
```

We immediately conclude that Chirpy is both a penguin and a bird. Does Chirpy fly? The rule for penguins rebuts the rule for birds, but the defeater for genetically altered penguins undercuts the rule for penguins. Since defeaters don't support their consequents, and since the other rules are either rebutted or undercut, d-Prolog responds 'can't tell' to the query `?- @@ flies(chirpy).`

4 Preempting defeaters

Notice that while the defeater undercuts the rule for penguins in the last example, it does not prevent the rule for penguins from rebutting the rule for birds. We may not want this to happen in cases where all of the rules involved are defeasible rules rather than defeaters. The most likely kind of situation where this could happen is where we have two Tweety Triangles involved. We'll begin with our southern Democrat rules:

```
democrat(X) :=  southern(X).
neg democrat(X) := conservative(X).
conservative(X) := southern(X).
```

These three rules constitute one 'Tweety Triangle'. Now we will add another.

```
democrat(X) := theatrical_agent(X).
neg democrat(X) := businessman(X).
businessman(X) := theatrical_agent(X).
```

These rules are at least plausible. All that is needed to complete our example is a southern theatrical agent.

```
southern(taylor).
tv_network_owner(tayler).
```

The question, of course, is whether Taylor is a Democrat. Looking at our knowledge base, it appears that the rules for southerners and for businessmen will rebut each other and that the rules for conservatives and for theatrical agents will rebut each other. However, the rule for conservatives is also rebutted by the more specific rule for southerners and the rule for

businessmen is also rebutted by the more specific rule for theatrical agents. It is reasonable to hold that a defeasible rule that is rebutted by a superior rule is no longer available to rebut other rules. In this case, we say that the defeasible rule rebutted by a superior rule is *preempted* as a rebutting defeater. Building this in to the system, we reach the intuitively correct conclusion that Taylor is a Democrat.

Like specificity, preemption of defeaters can be enabled or disabled. The query to toggle between the two states is

```
?- preempt.
```

In response to the query, d-Prolog informs the user whether preemption has just been enabled or disabled. By default, preemption is disabled when d-Prolog is loaded. Preemption has a fairly high computational cost and situations where it is needed to get intuitively correct results are rare.

5 Conflicts and NBF

The Nixon Diamond illustrates the most common way that the heads of two rules may be incompatible: when they are complements of each other. But this is not the only way. For example, One cannot be both a capitalist and a Marxist. In d-Prolog, we can represent this by the clause

```
incompatible(capitalist(X),
             marxist(X)).
```

With this clause in the knowledge base, the two rules

```
capitalist(X) := owns_restaurant(X).
marxist(X) := born_in(X,china).
```

become competitors. So rules are competitors if their heads are complements of each other or if there is a clause for the predicate `incompatible` which states that they are incompatible with one another. Rather than introduce the predicate `incompatible`, one might contemplate using two rules like

```
neg capitalist(X) := marxist(X).
neg marxist(X) := capitalist(X).
```

However, these two rules will cause looping in d-Prolog. Assume Ping, who was born in China, owns a restaurant. To show that Ping is defeasibly a capitalist using the rule

```
capitalist(X) := owns_restaurant(X).
```

we must determine that the rule is not defeated for Ping. To do that, we must show that the rule

```
neg capitalist(X) := marxist(X).
```

either is not satisfied or is defeated for Ping. Since Ping was born in China, we will have to show that the rule

```
neg marxist(X) := capitalist(X).
```

is satisfied for Ping. But this brings us back full circle to our original goal. We prevent this by building the use of the predicate `incompatible` into our defeasible inference engine.

The d-Prolog inference engine does not process cuts (`!`), disjunctions (`;`), and the built-in negation-by-failure (`not`) properly. By default, d-Prolog does not check for the occurrence of these functors and the behavior of the inference engine is unpredictable if they are encountered. However, the query

```
?- syntax.
```

enables the d-Prolog syntax check which will examine rules for the occurrence of these functors before the rule is used. If one of these functors is found, the user is alerted and d-Prolog aborts the current query.

While the use of `not` in d-Prolog programs is not allowed, d-Prolog still supports negation-by-failure. In fact, it allows the programmer to specify for which predicates the 'closed world assumption' is to be made. Suppose, for example, we want to make the closed world assumption for citizenship: unless there is evidence of citizenship, the query

```
?- @ neg citizen(X).
```

should succeed. To accomplish this, we add the presumption

```
neg citizen(X) := true.
```

to the knowledge base. In d-Prolog, any rule whose body contains at least one clause other than `true` is more specific than any presumption. So any positive evidence of citizenship will defeat our presumption.

6 The YSP

As a final example, we will take a look at the Yale Shooting Problem (Hanks and McDermott 1987). This will emphasize some important differences between d-Prolog and other nonmonotonic formalisms. Simplifying the example, an assassin waits with loaded pistol for her victim. When he arrives, she points the gun at his head and pulls the trigger. Assuming that pointing a loaded pistol at a persons head and pulling the trigger normally causes death, the question is whether the victim in the example winds up dead. To infer the victim's death, we must infer that the gun is loaded at the time the trigger is pulled. We need a 'frame' axiom to support this inference, and the YSP turns out to be a version of the frame problem.

McDermott (1987) proposed that nonmonotonic logic was the solution to the frame problem. We could formulate a single nonmonotonic axiom saying of every 'fact' that it tended to persist through time. McDermott rejected this solution in (Hanks and Mcdermott 1987) citing the YSP as a crucial example of why it doesn't work. The difficulty is that we will have two nonmonotonic rules, one saying that the gun tends to stay loaded and another saying that pointing a loaded gun at a persons head and pulling the trigger tends to cause death. McDermott's nonmonotonic formalism (McDermott and Doyle 1980) as well as McCarthy's circumscription (1980, 1986), Reiter's default logic (1980), and Moore's autoepistemic logic (1984), prescribe violating the smallest number of nonmonotonic principles possible. So we can violate either the the persistence principle or the causal principle for loaded guns, though it would be gratuitous to violate both. These two options produce two distinct fixed points, minimal models, or extensions for our theory. In the one where the gun becomes unloaded, the victim survives.

Defeasible logic does not generate multiple extensions. If it did, a backward chaining nonmonotonic inference engine based on it would not be possible.

The YSP is represented very simply in d-Prolog using McDermott's *situation calculus*.

```
true(alive(victim),s).
true(loaded(gun),s).
true(F,result(E,S)) := true(F,S).
true(dead(X),
     result(shoot_at(gun,X),S)) :=
        true(loaded(gun),S),
        true(alive(X),S).
incompatible(true(alive(X),S),
             true(dead(X),S)).
```

The first two clauses tell us that the victim is alive and the gun is loaded in the initial situation `s`. The first rule is our temporal persistence principle and the second rule is the causal principle for shooting people with loaded guns. The last clause says that it is impossible for anything to be both alive and dead in the same situation. The crucial queries are complex:

```
?- @ true(dead(victim),
          result(shoot_at(gun,victim),
          result(wait,s))).
?- @ true(alive(victim),
          result(shoot_at(gun,victim),
          result(wait,s))).
```

The atom `wait` represents the assassin's waiting for the victim. In d-Prolog, the first query succeeds and the second fails.

The reason we get the intuitive result is that d-Prolog backward chains through rules whose bodies involve times earlier than their heads. It is natural to write rules this way since we conceive of causation as moving forward rather than backward in time. The result is that defeasible rules applied at earlier times are less likely to be defeated than rules applied at later times. This applies particularly to the persistence principle.

The Yale Shooting Problem is an important test for a nonmonotonic formalism. Defeasible logic and d-Prolog have not problem with it.

7 Utility predicates

A number of utility predicates have been incorporated into d-Prolog to assist users in developing d-Prolog knowledge bases. I will discuss these briefly.

When Prolog reconsults a file, it reads the file into memory while eliminating any clauses already in memory for the predicates that occur in that file. Most Prolog implementations assume that all clauses for the same predicate will occur together in the file. If a file containing only the clauses

```
f(1).
g.
f(2).
```

is reconsulted while memory is empty, the result will be that only the clauses

```
g.
f(2).
```

will be in memory after the file is reconsulted. Because the clauses for the predicate `f` are separated by a clause for `g`, most Prologs will treat the second clause for `f` as if it is the first occurrence of a clause for `f`. The clause `f(1)` that was just stored in memory is eliminated before the clause `f(2)` is stored in memory. Thus, the Prolog programmer must be sure that clauses for the same predicate occur together in files that may be reconsulted. This behavior raises a special problem for d-Prolog. Consider the following set of clauses:

```
f(1).
f(X) :- g(X).
neg f(2).
f(X) := h(X).
neg f(X) :^ j(X).
incompatible(f(X),k(X)).
```

From the perspective of d-Prolog, each of these is a clause for the predicate `f`. But from the perspective of Prolog, only the first and second are clauses for `f`. The third is a clause for `neg`, the fourth is a clause for `:=`, the fifth is a clause for `:^`, and the last is a clause for `incompatible`.

d-Prolog's `reload` reconsults a file in an intelligent manner. It determines the predicates for clauses from the perspective of d-Prolog, and it remembers which predicates have been seen during the current reload operation. It will not discard clauses mistakenly even when clauses for the same predicate actually are separated in the file.

`reload` also builds an internal dictionary of all the predicates for which clauses are being added to memory. This dictionary is used for a couple of other functions. First, using `dictionary` as a query will produce a list of all predicates loaded using `reload` together with their arities. Second, using `contradictions` as a query will initiate a search for contradictory conclusions that can be derived from the clauses in memory. To perform this search, `contradictions` uses the internal dictionary to construct the contradictory queries that it tests. `reload` expects the file to be loaded or reloaded to have the extension `.dpl`.

With the d-Prolog clauses for the predicate `f` listed above, the query

```
?- listing(f).
```

will produce a list containing only the first two clauses, i.e., those clauses that from the perspective of Prolog have `f` as their predicate. The d-Prolog query

```
?- dlisting(f).
```

will produce a list containing all the clauses in the example.

The query

```
?- rescind(Predicate/Arity).
```

will remove from memory all facts, strict rules, defeasible rules, defeaters, and incompatibility statements that have `Predicate` as their predicate. The query

```
?- rescindall.
```

will rescind every predicate stored in the internal d-Prolog dictionary, thus eliminating the entire d-Prolog knowledge base from memory.

d-Prolog's `whynot` is an explanatory facility. In monotonic systems, there is always a single explanation for why a particular statement is not derivable: no rule supporting the statement is satisfied. But in a nonmonotonic system, a statement may not be derivable even though some rule supporting the statement is satisfied. In the case of the Tweety Triangle, we may want to know why the query

```
?- @ flies(tweety).
```

fails. We ask for an explanation with the query

```
?- whynot(flies(tweety)).
```

d-Prolog's response is that while the rule

```
flies(tweety) := bird(tweety).
```

is satisfied, so is the rule

```
neg flies(tweety) := penguin(tweety).
```

Furthermore, the first rule is not superior to the second rule. Thus, `whynot` explains how a rule has been rebutted or undercut.

8 Continuing work

An earlier version of d-Prolog was used in a system for selecting an appropriate forecasting method for a business (Nute *et al.* 1990). Many of the features described here were not included in that version. Several intermediate versions resulted from *ad hoc* changes to handle defeasible specificity, preemption, etc. The current version is a new implementation recently finished. It has not yet been used in any application although we have several domains in mind for development.

A proof generator for defeasible logic is also currently under development. This system not only tests to see whether queries succeed, but also constructs and displays a proof tree if the query succeeds. Finite defeasible theories without function symbols are decidable (Nute 1990). The proof generator actually incorporates a decision procedure for these theories.

References

Geffner, H. 1992. *Default Reasoning: Causal and Conditional Theories.* MIT.

Hanks, S. and McDermott, D. 1987. Nonmonotonic logic and temporal projection. *Artificial Intelligence* **33**:379–412.

Loui, R. 1987a. Response to Hanks and McDermott: temporal evolution of beliefs and beliefs about temporal evolution. *Cognitive Science* **11**:303–317.

Loui, R. 1987b. Defeat among arguments: a system of defeasible inference. *Computational Intelligence* **3**:100-106.

McCarthy, J. 1980. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence* **13**:27–39.

McCarthy, J. 1986. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence* **28**:89–116.

McDermott, D. 1987. We've been framed: or why AI is innocent of the frame problem. In Z. Pylyshyn (ed.), *The Robot's Dilemma: the Frame Problem in Artificial Intelligence.* Ablex Publishing Company, Norwood, NJ:113–122.

McDermott, D. and Doyle, J. 1980. Non-monotonic logic I. *Artificial Intelligence* **13**:41–72.

Moore, R. 1984. Possible-worlds semantics for autoepistemic logic. *Proceedings of the 1984 Non-monotonic Reasoning Workshop.* AAAI, Menlo Park, California.

Nute, D. 1991 A decidable quantified defeasible logic. *Proceedings of the 9th International Congress on Logic, Methodology, and the Philosophy of Science*, Uppsala, Sweden, August 1991, in press.

Nute, D. 1992. Basic defeasible logic. In L. Fariñas del Cerro and M. Penttonen (eds.), *Intensional Logics for Programming*, Oxford University Press:125–154.

Nute, D., R. Mann, and B. Brewer. 1990. Controlling expert system recommendations with defeasible logic. *Decision Support Systems* **6**:153–64.

Pollock, J. 1991. A theory of defeasible reasoning. *International Journal of Intelligent Systems* **6**:33–54.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* **13**:81–132.