EVID: A SYSTEM FOR INTERACTIVE DEFEASIBLE REASONING

ROBERT L. CAUSEY

JANUARY 1990 AI90-119

EVID: A System for Interactive Defeasible Reasoning

Robert L. Causey*

Department of Philosophy, and
Artificial Intelligence Laboratory
The University of Texas at Austin

Abstract

I describe a system for interactive, automated defeasible reasoning. An application program using this system can infer a conclusion defeasibly from a conjunction of supporting facts together with an appropriate general rule. This particular inference of the conclusion might be defeated by additional facts, although other independent evidence could still support the conclusion on the basis of other rules. The system has been implemented in Prolog, and includes an extensive logical interface that permits the user to interact with and override an application program's defeasible conclusions, subject to certain constraints on the user's consistency. Possible applications to decision support systems are described.

Keywords: Defeasible Reasoning, Default Reasoning, Nonmonotonic Reasoning, Decision Support Systems, Logic Programming.

**Mailing address:* Department of Philosophy, Waggener Hall 316, University of Texas, Austin, Texas 78712-1180 USA. *E-mail*: rlc@cs.utexas.edu

Some of the equipment and software for this research has been supported by U.S. Army Research Office Grant ARO-DAAG29-84-K-0060 to the University of Texas Artificial Intelligence Laboratory. I wish to thank Bruce Porter and Fletcher Mattox for assistance with setting up Quintus™ Prolog. I also thank Donald Nute for supplying his d-Prolog with sample programs and documentation. The implementation of the Arity™ version of EVID was independently supported and performed by the author.

1. Introduction to Defeasible Reasoning

1.1. Practical decision making often depends on information that is not directly available to us, but which can be inferred from other information that is available. For instance, while driving we may not directly experience the feel of a slick street, but infer that the street is slick from the visible evidence that it has just begun to rain. This inference is based on the rule-of-thumb that, in *typical* situations, if rain has begun to fall on a previously dry street, then that street is slick. There usually will be additional rules-of-thumb linking other kinds of evidence (e.g., an oil spill) to a given conclusion (e.g., slick street). Such rules may have exceptions; for instance, a special kind of road building material may prevent the onset of rain from causing slickness. Thus, the additional information that the particular street in question is made of this material will *defeat* (block) the *inference* of slickness from the evidence that it has just begun to rain.

In recent years there has been much interest in what is called "default", or "defeasible", or "nonmonotonic" reasoning. The first two terms have been used in a variety of ways by different writers, but it is generally the case that "default" reasoning is related to "defeasible" reasoning, and both of these are special types of "nonmonotonic reasoning". Many articles on the latter subject are reprinted in [4], and Chapter 6 of [3] is a useful exposition. In connection with decision support systems, defeasible reasoning is considered in [6], [5], and [1], where many other references to the literature are cited. Some comparisons with this literature will be made in Section 5.

1.2. To avoid confusion with other terminology in the literature, I will use the following general characterization: A *defeasible rule* is a special form of conditional (if-then) sentence[1] that permits one to infer a *conclusion sentence* (corresponding to the consequent of the rule) from a conjunction of supporting *evidential sentences* (corresponding to the antecedent), provided that this inference is not blocked (defeated) by other statements of *defeating conditions*. A conjunction of defeating conditions is called a *defeater*. There may be many defeasible rules with the same consequent following from various antecedents (conjunctions of different kinds of supporting evidence). In general, there will also be a variety of defeaters that block inferences of

[1]In formal logic, "rule" usually refers to a rule of inference, not to a sentence. However, the use of the term "rule" for if-then sentences is now customary in the Artificial Intelligence literature and I will follow this custom. To refer to a genuine rule of inference I will use "rule of inference".

instances of the consequent from one or more of the conjunctions of supporting evidence. Notice that a defeater does not defeat the rule; it blocks an inference that would otherwise be based on that rule together with supporting evidence. This characterization is still vague in some respects; the following simple, fictional example will illustrate the key ideas involved. It is assumed that the reader is familiar with first order predicate logic; some familiarity with Prolog programming will be helpful. In Prolog, predicate and individual constants are written in lower case, and variables in upper case. I will follow those conventions. In later sections, I will use bold face for Prolog code. The following examples are presented informally, but are also in bold face because I will later show how they can be implemented in Prolog.

Suppose that we are interested in whether or not a person uses a computer in his or her work. We introduce a predicate, **ucw(P)**, which is true of any person **P**, if **P** personally interacts with a computer as a part of **P**'s job. Such information might be of interest to a salesperson representing a computer manufacturing company. A person's occupation, together with certain other facts regarding that person, will serve as evidence that **ucw(P)**. For a simple example, we assume the following fictional, but not implausible, rules:

ucw(P) if(defeasibly) (engineer(P) & can_type(P)). (1)

ucw(P) if(defeasibly) (professor(P) & can_type(P)). (2)

The term **if(defeasibly)** will be given an exact interpretation in a later section, but its intended meaning, in the case of (1), can be roughly described as follows: The condition **(engineer(P) & can_type(P))** constitutes one piece of *prima facie* evidence for the conclusion **ucw(P)**. Thus, for a typical person **P**, who satisfies this *prima facie* evidence, we may infer **ucw(P)**. By "typical" I mean that we do not have any defeating conditions that would *defeat* (override) the evidential support of **(engineer(P) & can_type(P))** *for* the conclusion **ucw(P)**, and hence block the inference of the conclusion. If such a defeating condition (a defeater) becomes available, then it will defeat this conclusion *for* this evidence. It is important to understand that such a defeat does not imply that this conclusion is false, for it might still follow from some other, independent evidence. But if no other supporting evidence is available, and the evidence **(engineer(P) & can_type(P))** is defeated, then we are not warranted in making a *defeasible inference* that **ucw(P)**. Yet, as will be seen later, we may still choose to believe that it is true in particular circumstances where we have other, special reasons for this belief. It should already be clear that the exact meaning of **if(defeasibly)** is rather complex and subtle; this meaning will

be explicated further in Section 3.

Defeaters (conjunctions of defeating conditions) are components of *defeater rules*. Here are three examples:

ucw(P) is defeated_for (engineer(P) & can_type(P)) if company_president(P). (3)

ucw(P) is defeated_for (professor(P) & can_type(P)) if art_professor($\overline{P}$). (4)

ucw(P) is defeated if fears_computers(P). (5)

The relationship, **defeated_for**, occurs in (3) and (4), so these rules state conditions for *relative defeats*. Thus, (3) asserts that the additional information, **company_president(P)**, defeats the evidential support of **(engineer(P) & can_type(P))** *for* the conclusion **ucw(P)**. In this case, **company_president(P)** is a "conjunction" of one statement; **company_president(P)** is *the defeater corresponding to* the defeater rule (3). Similarly, if we know that **bob**, say, is a professor who can type, then by (2) we may defeasibly infer that **ucw(bob)**, but if we find out that **bob** is also an art professor, then this inference is defeated by (4). It is important to understand that it may still be the case that **ucw(bob)**, but we are no longer *entitled to infer* this by means of (2) when (2) has been relatively defeated. The conditions, **company_president(P)** and **art_professor(P)**, are *relative defeaters* because each of them defeats (blocks) an inference based on (relative to) some piece of evidence. If **bob** is an engineer, can type, and is also a professor, then an inference from either (1) or (2) might be relatively defeated by either (3) or (4) together with the corresponding relative defeater, respectively, while the other inference is still allowable.

The condition, **fears_computers(P)**, in (5), is an *absolute defeater*. If **fears_computers(bob)** is true, then we are not allowed to infer that **ucw(bob)** from *any* evidence. In other words, **fears_computers** defeats any supporting evidence corresponding to the antecedent of any defeasible rule with the conclusion **ucw(bob)**. Yet, as will be seen later, we may still choose to believe that **ucw(bob)** in special circumstances where we have other, special reasons for this belief.

1.3. In order to summarize the ideas introduced above, let **concl**, $\mathbf{ev_1}$, $\mathbf{ev_2}$, ..., $\mathbf{ev_k}$, be predicates followed by zero or more free variables or individual constants. (If there are no variables or constants, the predicate is treated as a propositional constant.) Then a defeasible rule will usually have the form given by this schema:

concl if(defeasibly) $(\mathbf{ev_1}$ **&** $\mathbf{ev_2}$ **& ... &** $\mathbf{ev_k})$**.** (6)

Later we will also allow negations of the $\mathbf{ev_i}$, using Prolog's negation by failure. As mentioned previously, the exact interpretation of **if(defeasibly)** will also be given later. I am not suggesting that it must be a new kind of if-then logical connective.

Now let $\mathbf{d_1}$, ..., $\mathbf{d_n}$, also be predicates followed by zero or more free variables or individual constants. Then a relative defeater rule has the form:

concl is defeated_for $(\mathbf{ev_1}$ **&** $\mathbf{ev_2}$ **& ... &** $\mathbf{ev_k})$ **if**
$(\mathbf{d_1}$ **& ... &** $\mathbf{d_n})$**.** (7)

An absolute defeater rule has the form:

concl is defeated if $(\mathbf{d_1}$ **& ... &** $\mathbf{d_n})$**.** (8)

The conjunctions in the antecedents of (7) and (8) are *relative* and *absolute defeaters*, respectively. The relative defeater rule schema (7) uses the 2-ary metapredicate **defeated_for**, whereas (8) uses the 1-ary metapredicate **defeated**. All of these characterizations will be refined later.

Defeasible rules, defeater rules, and both relative and absolute defeaters, are used in practical reasoning, including the reasoning required for decision making. Some might insist that such rules should be replaced by assertions of conditional probability. I would agree that probabilistic decision models are often useful, when suitable probability values can be obtained. But suitable values are often not available. In fact, most of our practical reasoning does not involve probability calculations; rather, it is based on nonprobabilistic, defeasible reasoning. Moreover, as pointed out in [6], pp. 98-99, a decision support system (DSS) that relies on probability values can be very difficult to expand and maintain.

It should be noted that many previous investigations of defeasible and default reasoning do not make use of the distinction between relative and absolute defeaters. In these investigations, and in the reasoning systems based on them, one usually finds a facility for handling only those

cases approximately corresponding to our absolute defeaters. J. Pollock's work is one exception to this; see [8], which distinguishes ''rebutting'' from ''undercutting'' defeaters. Although there are significant differences between Pollock's approach and the present one, his distinction seems to correspond to the current distinction between absolute and relative defeaters. The above example and the further discussion below should make it very clear that such a distinction is needed for adequate representation of practical, defeasible, evidence-based reasoning.

2. Functional Requirements for an Interactive Defeasible Reasoning System

2.1. Statements (1)-(5) in the previous section are examples of ''domain knowledge'' because they make assertions about a particular application domain. Let us call such knowledge ''long-term'' if it is stored in a computer program and we do not expect that it will require revision very often. An *interactive defeasible reasoning program* (IDRP) is an interactive program containing long-term knowledge, that is used together with user-entered ''current data'' to provide advice to the user. For instance, a user query might ask whether **ucw(bob)**, and the response will be inferred from the long-term knowledge of the IDRP together with user-supplied, current information about **bob**. Although the general concept of a DSS is broader (including probability based systems), it should be clear that an IDRP could be used as a DSS or as part of one.

This article will not present a general theory of defeasible reasoning, but it will describe some of the most important features of an IDRP that are required to assist with practical decision making. In particular, we will be concerned with IDRP's that are able to perform evidence-based reasoning, and are designed so that the user has the ability, subject to consistency of his own inputs, to defeat (block) the defeasible inferences of the program. A number of other highly desirable features of an IDRP will also be described. These conditions will first be summarized in general terms below; later I will show how these features can be realized within the backward chaining deduction environment of Prolog.

Of course, one does not want to program from scratch each particular IDRP for each application of interest. It is more convenient to have a general ''shell'' that interprets domain knowledge and assists user interaction. Such a shell will be called an *interactive defeasible reasoning system (IDRS)*. For the discussion in the remainder of this section, let us give the name EVID to an idealized IDRS. I will describe some of the main functional characteristics

which EVID should have. In some cases it will be convenient to use descriptions that directly refer to features of EVID; in other cases, it will be more convenient to describe how an IDRP interpreted by EVID should behave. In the latter case, however, this behavior will still largely be the result of the general functionality of EVID.

2.2. Before we proceed, it is important to note that an IDRP will usually contain some *nondefeasible* rules in addition to its defeasible ones. For example, we might have

professor(P) if art_professor(P). (9)

The next few paragraphs describe desirable functional requirements of the ideal EVID.

2.2.1. If the user enters the information, **art_professor(bob)**, the IDRP will be able to infer **professor(bob)** from (9). Thus, some of the system's conclusions will hold *positively* (as in this case), and some will only hold *defeasibly*, as we have already seen. EVID should be able to tell the user how each of the IDRP's conclusions holds.

2.2.2. If the IDRP can infer a conclusion from one or more pieces of evidence, for instance **ucw(bob)** from **(engineer(bob) & can_type(bob))**, the system should be able to *justify* this conclusion by citing *all* of the supporting evidence.

2.2.3. Sometimes the user will expect the IDRP to infer a conclusion, but it does not. Often this will happen because all available supporting evidence has been defeated. The system should be able to report these defeats to the user.

2.2.4. The user may want a complete deduction of one of the system's conclusions. EVID should contain a proof-trace facility for displaying such proofs.

2.2.5. In some (perhaps most) applications there will be times when the user wants to know what additional user-input data would enable the IDRP to infer a conclusion which it currently cannot. For instance, suppose the IDRP does not yet have any information about **bob**, but the user wants to know what would enable it to infer **ucw(bob)**. EVID should be able to analyze the domain rules and tell the user what pieces of evidence (if entered) would lead to the conclusion **ucw(bob)**.

2.2.6. There will be times when the system infers a defeasible conclusion which the user doubts. In some cases of this type the user may suspect that there is additional defeating information available to him that has not been entered into the system, but he may not know exactly what additional information is relevant in this manner. EVID should be able to tell the

user what additional data would serve as either relative or absolute defeaters.

2.2.7. Suppose that the IDRP infers from appropriate evidence that, defeasibly, **ucw(bob)**. Suppose further that the user does not have available any additional data that would defeat this inference. Still, the user may have good reasons for believing that, *in this particular situation*, the available evidence is not sufficient to conclude that **ucw(bob)**. Indeed, the user may have good reason for believing that there is *no evidence of any kind* to warrant the conclusion that **ucw(bob)**. Regardless of how thoroughly the long-term knowledge of the IDRP has been developed, there will be atypical situations that are not covered by suitable defeater conditions. Thus, occasionally the IDRP may defeasibly infer some conclusion with which the user disagrees, and such that the user cannot defeat this conclusion simply by the input of additional information known to the user. *EVID should allow the user to defeat this conclusion, either relative to particular supporting evidence, or absolutely, provided that the user does not contradict himself in so doing*. Thus, subject to self-consistency, EVID should permit the user to have ''last word'' on defeasible conclusions. Such a user-entered override of an IDRP conclusion is called a ''user defeat''.

2.2.8. The user is permitted to add new information and to defeat some of the system's conclusions. EVID should also allow the user to remove data previously added by the user and to undefeat conclusions that the user has previously defeated (either relatively or absolutely). Naturally, there is the risk that the user may, in performing various combinations of such actions, either generate a logical contradiction from his inputs, or create a conflict that arises from specific domain knowledge. Also, the user may introduce certain kinds of redundancies which, although not logically harmful, may create deductive inefficiencies. EVID should be designed to prevent, as far as is practical, the user from performing such undesirable actions, while still leaving the user ''reasonable freedom'' in ''having the last word.'' It is difficult to characterize this general aim precisely and, because of computational limitations of logic, impossible to check for all inconsistencies. Therefore, these constraints on user interactions must be largely formulated as particular implementation features (which will be described in a later section).

The previous paragraphs describe, very generally, the functionality desired in our idealized EVID, an interactive defeasible reasoning system. Of course, many details still need to be explicated, and there are other, special features that either are intrinsically desirable, or are useful in achieving the general functionality we desire. The following sections give more detail in the context of a particular Prolog implementation.

```
  % Relative defeater rules.....
defeated_for((engineer(P), can_type(P)), ucw(P)) :-
     engineer(P),
     can_type(P),
     company_president(P).

defeated_for((professor(P), can_type(P)), ucw(P)) :-
     professor(P),
     can_type(P),
     art_professor(P).


  % Absolute defeater rule.....
defeated(ucw(P)) :- fears_computers(P).
```

First of all, notice that some of the predicates in the example are declared to be **definite** and others **defeasible**. EVID's own internal predicates are declared (within EVID) to be **evid_predicates**; any predicate in an application program (such as the above example) that is neither an **evid_predicate** nor a built-in Prolog system predicate must be declared as **definite** or **defeasible**. EVID makes extensive use of these declarations on frequent occasions of runtime predicate type checking. Although runtime checking uses extra computations, the EVID declarations enhance program efficiency and error checking in other ways. In the following, a predicate that is either definite or defeasible is called a *domain predicate*.

EVID declarations were actually invented, however, for epistemological reasons. It is assumed that some predicates are used to represent ''basic facts'' while others are used to represent ''defeasible conclusions'' that are (usually) inferred from the basic facts. The user is given complete freedom to decide which predicates are definite and which are defeasible, and the user's declarations will depend on the features of the particular application domain under consideration. In extreme situations, all predicates might be defeasible or all definite (although in the latter case, the knowledge base (KB) will not have any defeasible rules). It would be out of place here to attempt a detailed epistemological justification for the EVID declarations. These declarations impose a constraint on the knowledge representation style of EVID-based IDRP's. However, it should become clear that the declarations have a natural motivation, and are useful in developing KB's and for program efficiency.

In the case of the above example, we assume that information such as **art_professor(jim)** is basic in the sense that it must be supplied to the program as an externally obtained fact. Hence, **art_professor** is declared to be definite. The predicate **professor** is also declared definite for the same reason. In addition, a fact such as **professor(jim)** can be inferred by the program from

art_professor(jim) by a nondefeasible rule, as shown in the program. On the other hand, **ucw** is declared defeasible since the program will be able to use one or more defeasible rules to infer facts such as **ucw(jane)** from other, current information in the program. In general, atomic facts stated in terms of defeasible predicates (such as **ucw(jane)**) are those which an IDRP can infer from *prima facie* evidence by inferences that can be blocked when additional, defeating information is obtained. Yet, the definite/defeasible distinction is not an absolute one, but is relative to the context of an application and determined by the designer of the KB.

To simplify the terminology, an atomic formula or the negation of an atomic formula (using Prolog's **not**, negation by failure) will be called a *literal*. If an atomic formula uses a definite or defeasible predicate, then that formula will also be called *definite* or *defeasible*, respectively. The sample application program contains a nondefeasible rule and several defeasible rules. A *nondefeasible rule* always has the form of a standard Prolog rule, with a definite formula in the head of the rule. The body of such a rule is always a conjunction of one or more literals. These literals can be formed from domain predicates or Prolog system predicates (such as those corresponding to numerical operations).

A *defeasible rule* must have a defeasible atomic formula in its head, and there are only two allowable forms of defeasible rules:

concl :-
    ev_1, ev_2, ... , ev_k,
    not(defeated_for((ev_1, ev_2, ... , ev_k), concl)). (10)

concl :-
    not(defeated(concl)). (11)

In (10) and (11), **concl** and the **ev_i** may have zero or more free variables or individual constants. The head, **concl**, is a defeasible atomic formula. The **ev_i** may be any literals using domain predicates or Prolog system predicates. Important distinguishing features of these two rule forms are the required uses of the evid_predicates, **defeated_for** and **defeated**. The reader should notice that the general form of (6) is implemented by (10) and interpreted by EVID in the form of (10). It is important to observe that both (10) and (11) are standard Prolog rules; they simply have special forms using the **defeated_for** and **defeated** metapredicates, and are further constrained by the types of predicates allowed to occur in particular parts of the rules. A rule of form (10) enables the program to infer **concl** from the evidence (**ev_1, ev_2, ... , ev_k**) provided that this inference of the conclusion is not **defeated_for** this particular evidence. Rules of form (11)

are rare, but can be used to infer that **concl** holds *prima facie* without any special evidence to support it, unless this conclusion is absolutely defeated.

Finally, the little sample program contains some relative defeater rules and an absolute defeater rule. Rules of these types have the forms:

defeated_for((ev_1, ev_2, ... , ev_k), concl) :-
ev_1, ev_2, ... , ev_k,
d_1, d_2, ... , d_n. (12)

defeated(concl) :-
d_1, d_2, ... , d_n. (13)

Rule forms (12) and (13) are the EVID-style implementations of (7) and (8), respectively. In (12) and (13), the $\mathbf{d_i}$ are the conjuncts of the relative and absolute defeaters, respectively; they may be literals formed from domain predicates or Prolog system predicates. In (12) the evidential conditions, $\mathbf{ev_i}$, are restated in the body of the rule so that Prolog will apply the relative defeat only to domain objects that satisfy these evidential conditions. Relative and absolute defeats are conceptually related, and EVID represents this relationship. EVID has an internal rule that says that a conclusion will be defeated for any evidence whatsoever, if that conclusion is (absolutely) defeated. Let **ev** abbreviate some conjunction of evidential conditions. Note that **defeated_for(ev,concl)** can be true while **concl** is also true, because there may be other, undefeated evidence that supports **concl**. If all currently true supporting evidence for **concl** is defeated, then a query of **concl** will return "no", but all this means is that Prolog cannot currently infer the conclusion. Additional, new evidence might result in **concl** again being inferable. If **defeated(concl)** results from current facts plus the IDRP's internal rules, then there can be no additional evidence that would enable the IDRP to infer **concl**, but even in this case EVID will (usually) permit the user to add **concl** to the current body of facts (as the next section will show). In other words, **concl** is logically consistent with **defeated(concl)**, since the latter only blocks the IDRP from inferring **concl**.

4. What EVID Does

4.1. Introduction to EVID's Operations. As previously mentioned, EVID is a large program with many predicates. In this article, I will only mention a few of the main EVID predicates that are primarily for user interactions. In order to use EVID, it is loaded into Prolog along with one or more application files. Suppose that EVID is loaded together with the sample application program listed in the previous section. One important EVID predicate is **addit**; it allows the user to assert new facts as current data for the application IDRP, but only after checking many things, some of which will be mentioned later. Initially, if we do **addit(engineer(bob))** and **addit(professor(bob))**, EVID will permit these facts to be asserted and will also cause them to be remembered by the system as **user_added** facts.

EVID has a predicate, **holds**, such that **holds(How,Sent)** informs us **How** a true sentence holds. **Sent** is either a definite or defeasible atomic formula constructed from a domain predicate followed by individual constants and variables. As is customary, free variables are assumed to be universally quantified, so a formula with free variables is actually a sentence. There are several possible values for how a sentence can hold. If the sentence is defeasible (i.e., is atomic with a defeasible predicate), then it *always* holds *defeasibly*. If the sentence is definite (is atomic with a definite predicate), and all derivations of this sentence use nondefeasible rules with no negation by failure, then the sentence holds *positively*. Currently, EVID also distinguishes two other values for **holds** that are intermediate between these cases, but these details will not discussed here.[4] In our current example, if we now ask the system, **holds(How,professor(bob))**, it replies "**How = positively**". If we ask, **holds(How,ucw(bob))**, it replies "**How = defeasibly**".

EVID includes a predicate **why**, which returns a special type of justification for conclusions. Since **ucw(bob)** is now true, we can ask why. With the output in a slightly revised format, here is what we obtain[5]:

---

[4]These modes apply to atomic formulas with definite predicates. A sentence of this form may have several derivations. Some of these derivations may depend on an ancestor premise that holds defeasibly. In other cases, there may be no defeasible ancestors, but there may be a derivation with an ancestor premise using Prolog's **not**, and hence negation by failure. The **holds** predicate is defined in such a manner that these intermediate cases are also distinguished.

[5]All of the sample output is copied directly from use of the programs, but it has been reformatted to save space and remove inessential features such as screen scroll pauses.

?- why(ucw(bob)).
ucw(bob)
HOLDS defeasibly.
IT IS CURRENTLY SUPPORTED BY THE FOLLOWING EVIDENCE:
[engineer(bob),can_type(bob)]
[professor(bob),can_type(bob)]

AND ALSO THE:
[SUPPORTING EVIDENCE ==>]
[IS DEFEATED BY ==>]

Notice that currently there is no evidence for **ucw(bob)** that is defeated, as is indicated by the blanks after the arrows. However, we might want to know what additional facts *would* defeat this conclusion *if* these facts *were* true. EVID has a predicate, **howdefeatit**, which tells us the following:

?- howdefeatit(ucw(bob)).
ucw(bob)
CAN BE DEFEATED BY ADDITION OF ANY OF THESE ABSOLUTE DEFEATERS
[fears_computers(bob)]

IT CAN BE DEFEATED BY ADDITION OF ALL OF THESE RELATIVE DEFEATERS
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[BY THE DEFEATER ==> ,company_president(bob)]
[SUPPORTING EVIDENCE ==> ,professor(bob),can_type(bob)]
[BY THE DEFEATER ==> ,art_professor(bob)]

CURRENTLY THE
[SUPPORTING EVIDENCE ==>]
[IS DEFEATED BY ==>]

Notice that **howdefeatit** gives the user considerable information. It first lists all of the absolute defeaters. It then lists all relative defeaters that are not presently true, but which would affect the current supporting evidence if they were. Finally, like **why**, it lists all current defeaters, if any. If it had been the case that **bob** were only a typing engineer, then **howdefeatit** would not have listed the **art_professor(bob)** defeater case.

Suppose that we now do **addit(company_president(bob))**. After this **howdefeatit** responds with:

?- howdefeatit(ucw(bob)).
ucw(bob)
CAN BE DEFEATED BY ADDITION OF ANY OF THESE ABSOLUTE DEFEATERS
[fears_computers(bob)]

IT CAN BE DEFEATED BY ADDITION OF ALL OF THESE RELATIVE DEFEATERS
[SUPPORTING EVIDENCE ==> ,professor(bob),can_type(bob)]
[BY THE DEFEATER ==> ,art_professor(bob)]

CURRENTLY THE
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[IS DEFEATED BY ==> ,company_president(bob)]

Notice that the now defeated supporting evidence (**engineer(bob), can_type(bob)**, together with its currently true relative defeater **company_president(bob)** have been moved from above to below the dashed line in the display. If we also **addit(fears_computers(bob))**, then we will obtain:

?- howdefeatit(ucw(bob)).
ucw(bob)
IS NOT TRUE, AND CURRENTLY THE
[SUPPORTING EVIDENCE ==> ,*all_evidence*]
[IS DEFEATED BY ==> ,fears_computers(bob)]
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[IS DEFEATED BY ==> ,company_president(bob)]

We now have both a relative and an absolute defeater active. The absolute defeat is indicated by ***all_evidence***, which is an EVID propositional constant which is declared to be definite and always holds positively. If a conclusion (such as **ucw(bob)**) is defeated for ***all_evidence***, then EVID will insure that *no* evidence and defeasible rules will support an inference of this conclusion. If we ask **why(ucw(bob))**, we will be told that **ucw(bob)** is not true and be advised to try **whynot**. The latter will give us the same information that **howdefeatit** shows here.

Suppose that someone had not seen the above displays, but queried the system, in its current state, about **ucw(bob)**. He would receive the response "no". Suppose this person wants to learn how this conclusion might be obtained. He could use the **howgetit** predicate as follows:

```
?- howgetit( ucw(bob) ).
ucw(bob)
IS DEFEASIBLE, AND IS CURRENTLY NOT TRUE.  WHEN TRUE AND
UNDEFEATED, THE FOLLOWING SETS OF EVIDENCE SUPPORT OR DENY IT
[computer_artist(bob)]
[engineer(bob),can_type(bob)]
[professor(bob),can_type(bob)]

CURRENTLY THE
[SUPPORTING EVIDENCE ==> ,*all_evidence*]
[   IS DEFEATED BY ==> ,fears_computers(bob)]
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[   IS DEFEATED BY ==> ,company_president(bob)]
```

Although not previously mentioned, there are occasions when one uses a rule that supports the *failure* of a conclusion, rather than its truth. The response of **howgetit** reflects this possibility without specifying which evidence supports affirmatively or negatively. Evidence for a denial of a conclusion normally contains the Prolog propositional constant **fail**, so it is fairly easy for the user to use other EVID predicates in order to distinguish the affirmative from the denying evidence.

4.2. User Overrides. The above examples show some of the basic ways in which EVID interprets an application program and acts as an informative interface between this program and the user. The examples show that the user can add additional facts to the application program and thereby (nonmonotonically) affect the conclusions this program can infer with EVID's assistance. Yet, when the user merely adds new factual information, he (or she) at most *indirectly* affects the system's inferences. In addition to such indirect effects, EVID also provides for *direct* user overrides of most of the program's defeasible conclusions and of its defeats of conclusions.

The latter type of override is simpler. Suppose that our example program was given just the following facts about **bob**: **engineer(bob), can_type(bob), company_president(bob)**. Then **ucw(bob)** does not hold because the inference of this conclusion is defeated by the information that **bob** is a company president. Presumably the user who entered these facts believes that they are all true. Also, we assume that the rules in the system are correct in the sense that they represent good information about **engineer, ucw**, etc. Yet, *in this particular instance* the user has other good, external reasons for believing that **ucw(bob)** is true *even though* **bob** is also a company president, and this user also wants the program "to share this belief" with him. In

such a situation the user simply uses **addit** to assert that **ucw(bob)**. Then this fact will be both relatively defeated and also true, as shown here:

?- why(ucw(bob)).
ucw(bob)
HOLDS defeasibly.
IT IS CURRENTLY SUPPORTED BY THE FOLLOWING EVIDENCE:
[it_is_a_defeasible_user_added_fact]

AND ALSO THE:
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[IS DEFEATED BY ==> ,company_president(bob)]

The reader should notice that **howgetit** and **howdefeatit** did not include the cases of direct user additions or overrides in their displays, since it is assumed that the user will know that these cases are almost always allowable. Note that **holds** considers that a user added fact holds defeasibly if it is built from a defeasible predicate, in this case **ucw**. Also, note that the only ''evidence'' applicable in this situation is that shown above (a user added fact). This affects the operation of **howdefeatit**; a query of **howdefeatit(ucw(bob))** will now remind the user that **ucw(bob)** is user added and should just be *removed.* Any user added fact can be removed by using the **removeit** EVID predicate. Finally, a user addition (using **addit**) would be allowed whenever **ucw(bob)** is not currently true, provided that it is not true because either there is no evidence in the system for it, or because any available evidence has been defeated by one or more relative or absolute defeats *as a result of the program's internal rules and data.* More generally, **addit** does not permit the user addition of a fact that is already true, regardless of how it happens to hold. As will be seen below, **addit** has other restrictions in the cases of user defeats.

Not only can the user override a program defeat by the addition of a fact (as just shown), he can also override a program defeasible inference by a manual defeat. EVID has two predicates, **defeatit_for** and **defeatit**, for this purpose. Suppose that the example program was given just the following facts about **bob**: **engineer(bob), professor(bob).** Then **ucw(bob)** holds defeasibly on the basis of two different sets of supporting evidence, and, as was shown previously, a query of **howdefeatit(ucw(bob))** will display one absolute and two relative defeaters. Yet, the user will know about direct user overrides, and one possibility is a relative defeat such as **defeatit_for((engineer(bob), can_type(bob)), ucw(bob))**. After this is done, we can observe this:

?- howdefeatit(ucw(bob)).
ucw(bob)
CAN BE DEFEATED BY ADDITION OF ANY OF THESE ABSOLUTE DEFEATERS
[fears_computers(bob)]

IT CAN BE DEFEATED BY ADDITION OF ALL OF THESE RELATIVE DEFEATERS
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[BY THE DEFEATER ==> ,company_president(bob)]
[SUPPORTING EVIDENCE ==> ,professor(bob),can_type(bob)]
[BY THE DEFEATER ==> ,art_professor(bob)]

CURRENTLY THE
[SUPPORTING EVIDENCE ==> ,engineer(bob),can_type(bob)]
[IS DEFEATED BY ==> ,user_defeated_for((engineer(bob),can_type(bob)),ucw(bob))]

A relative user defeat of this kind would be appropriate if the user has good, external reasons for believing that, *in this particular instance*, the fact that **bob** is an engineer who can type is *not* sufficient evidence for inferring that **ucw(bob)**. In effect, the user is telling the program to ignore, in the case of **bob**, this particular supporting evidence, while letting the program make inferences based on other, nondefeated evidence. If the user is really convinced that he knows **bob** so well that there is *no* defeasible evidence whatsoever that would convince him that **ucw(bob)**, then the user can do **defeatit(ucw(bob))**, which defeats the conclusion for all evidence. If this is done, **howdefeatit(ucw(bob))** will report that **ucw(bob)** is not true and has been user defeated for ***all_evidence***. The query **whynot(ucw(bob))** returns a similar report.

It was mentioned above that one may use **addit** to add a fact to the system if that fact is not currently true in the KB because there is no undefeated supporting evidence for it, *as a result of the program's internal rules and data.* However, if the fact is not currently true as a result of the use of **defeatit**, then it has been *defeated by the user for all evidence.* In such a situation, **addit** will not permit the user to add in this fact. If the user wants it to be true again, he must first use the **undefeatit** predicate. This restriction on **addit** does not apply if the user has only made *relative* user defeats.

5. Discussion of EVID's Design

5.1. EVID's Logical Interface. Sections 3 and 4 describe how defeasible rules are formulated in an application program and how EVID interprets these rules and provides a user interface to the application program. This description shows that the current EVID program satisfies the general functional requirements that are specified in Section 2[6]. I will now present more detailed motivation for some of these functional requirements.

By "user interface" I am not referring to such things as windows, icons, and menus, but rather to what might be called a *logical interface*. EVID predicates such as **holds, why, whynot, howgetit**, and **howdefeat** give the user information about various logical relationships between evidence, conclusions, and defeating conditions. EVID contains other predicates that provide additional information about possible defeaters and possible evidence, and about all currently true evidence and defeaters in the system, etc. Furthermore, predicates such as **addit, removeit, defeatit_for, undefeatit_for**, etc., permit the user to make interactive changes in the *epistemological state* of the system while they perform certain logical checks and impose some logical constraints on these interactive changes. Thus, it is appropriate to say that EVID provides a *logical interface* between the application program and the user.

This logical interface is considerably more complex than the above simple examples indicate. The complexity results largely from the major design decision to give the user "the last word," subject to the user's own "self-consistency," when the user disagrees with either a defeasible conclusion or a defeat of a conclusion inferred from the application program. "Self-consistency" as used here does not merely mean strict logical consistency, but refers more broadly to the avoidance of epistemically irrational actions.

In addition, some other constraints are imposed in order to avoid useless redundancies and for the sake of efficiency. The extent of these interface complexities is clear from an examination of the program. For instance, in the current implementation, the specification of the **defeatit_for** predicate uses eighteen Prolog clauses, many of which call other complex predicates. I will not attempt to describe the interface in detail, but will instead mention some of its additional, salient features.

One design feature is to prevent the user from asserting "irrelevant facts", to which no

---

[6]In addition to the features described in Sections 3 and 4, EVID also has a proof-trace facility as mentioned in Section 2.2.4.

program rule would apply. Thus, **addit** does type checking and will not allow the user to add a fact that is neither definite nor defeasible. Also, it will not allow the addition of any fact that is already true in the program, either as a direct assertion or indirectly by inference. This restriction has several advantages, including program efficiency, but its primary motivation is the following: An application program is intended to be a kind of friendly, knowledgeable adviser to the user, with EVID as their interface. The user gives the program some *external facts* (perhaps about a particular decision problem) that the program does not have, and then the program infers conclusions from these facts for the user's information. These external facts supplied by the user are of two main kinds: basic data that the program is unable to infer even without any relevant defeating conditions being true, or conclusions for which the program has only defeated evidence. In either situation, the fact the user wishes to add is not currently true in the program. Hence, there is no need to allow the user to add facts that are already true, and concern for efficiency motivates blocking such additions.

Suppose that the program has defeasibly inferred **concl**, but the user is convinced that no evidence should be allowed to support this particular conclusion. Then, she (the user) can do **defeatit(concl)** and thereby defeat this conclusion for all possible evidence. As previously mentioned, in such a case **addit** will not permit her to assert **concl** back into the current state of the program's KB. Unlike the cases just mentioned, in this one the fact, **concl**, is not true in the program, but the reason for this is the user's own action of defeating it. Thus, the user believes that there is no possible evidence to support **concl**, so if she asserted it, she would be asserting something for which she already believes there can be no evidential justification. The **addit** predicate prevents her from doing this.

The **defeatit_for** predicate is 2-ary and needs to check for more potential problems than the **addit** predicate. Suppose that the user enters **defeatit_for(ev,concl)**, where **ev** is a conjunction of one or more atomic formulas. Then **concl** must be a defeasible atomic sentence, and the entire conjunction **ev** must be of the appropriate form to be a possible piece of evidence for **concl**. The latter condition implies that the program must have at least one rule of the proper form relating **concl** to **ev**. This requires checking a number of aspects of these formulas. Also, additional checks apply in the special case where **ev** is ***all_evidence***. This case arises automatically in EVID when using the **defeatit** predicate for absolute user defeats. Also, **defeatit_for** does not permit a relative or absolute defeat of **concl** whenever **concl** is user added, so additional checks are required to obtain this constraint.

Application programs sometimes have rules that lead to a more complicated situation than has previously been mentioned. Suppose that we have an application program that recommends how a person is likely to travel from location A to location B under certain conditions. Such a program might be useful, for instance, in military intelligence contexts. If the travel is over an island, say, there might be options of travel by land (by means of various roads), travel by air, or travel by sea. This program could have a rule that states that a person travels by air on a date if (defeasibly) that person flies something (jet, balloon, etc.) on that date. Let us assume that **travels_by** (land, air, sea) and **flies** are defeasible and occur in other rules. Yet, in this case, the connection between these predicates is assumed to be very strong in the sense that one cannot both fly and not travel by air. We want the user to be able to defeat the program's defeasible conclusions but, because of this strong connection, a defeat of ''travels by air'' should also imply a defeat of ''flies''. In abstract form, here is a more extended example of these ideas:

```
definite(ev(_)).
defeasible(concl0(_)).
defeasible(concl1(_)).
defeasible(concl2(_)).
defeasible(concl3(_)).

concl0(X) :-
  ev(X),
  not( defeated_for( ev(X), concl0(X) ) ).

concl1(X) :-
  concl0(X),
  not( defeated_for( concl0(X), concl1(X) ) ).

concl2(X) :-
  concl1(X),
  not( defeated_for(concl1(X),concl2(X)) ).

concl3(X) :-
  concl2(X),
  not( defeated_for(concl2(X), concl3(X)) ).

 % Contra-defeaters...
defeated(concl0(X)) :- defeated(concl1(X)).
defeated(concl1(X)) :- defeated(concl2(X)).
defeated(concl2(X)) :- defeated(concl3(X)).

ev(bob).
```

The last three rules are *contra-defeaters*, which can hold when the evidence for one

defeasible conclusion is another defeasible formula. Contra-defeaters are used when we want to guarantee that an absolute defeat of the conclusion also leads to an absolute defeat of the evidential formula, as in the example of travels by air and flies. There can also be extended chains of contra-defeaters, as the above little program illustrates. If this program is interpreted by EVID, initially **concl0(bob), concl1(bob), concl2(bob), concl3(bob)** all hold defeasibly. If the user does not accept the last conclusion, she may do **defeatit(concl3(bob))**, which produces a user defeat of this conclusion for ***all_evidence***. In addition, because of the contra-defeater rules, **concl0(bob), concl1(bob), concl2(bob)** are also defeated. It was explained above that, since **concl3(bob)** is user defeated, **addit** will not permit the user to add this conclusion back into the program. But **concl0(bob), concl1(bob), concl2(bob)** are also defeated for all evidence by the user, indirectly, because of the contra-defeater chain. Thus, **addit** will also not permit the user to add any of these back into the program's current KB. If the user does **addit(concl0(diane))** then, because of the contra-defeater chain, she is similarly not permitted by EVID to apply **defeatit** to any of the other numbered conclusions that follow from it. As might be guessed, checking for contra-defeater chains is computationally rather expensive but, fortunately, long chains seem to be uncommon in practical applications. Predicates in EVID's logical interface, including **addit** and **defeatit_for**, also check for other situations, but these are more specialized and will not be discussed here.

I believe that user interaction will be very important in many DSS's. There has already been much investigation of nonmonotonic reasoning systems, but little attention has been given to user interaction. Although still somewhat sketchy, I hope that the above description has persuaded the reader that the goal of effective user interaction leads to important and difficult problems regarding the design and implementation of a logical interface. In particular, when the user is permitted to defeat evidential support for conclusions and also add new facts to the program, it becomes difficult to characterize exactly what we should mean by "user self-consistency." When a number of pragmatic and epistemological issues are considered, such as the complications of contra-defeater chains, this characterization problem becomes even more complex. In the design of EVID I have been guided by experimental testing of a number of different types of logical problems and application programs (see Section 6). EVID does work well with many applications, but it will undergo further development.

5.2. Some Comparisons. Although I want to avoid extended comparative discussions, a few points are worth noting here. First of all, as far as I know, no previous theoretical discussion or practical implementation of any nonmonotonic reasoning system has included any detailed investigation of the logical interface requirements for safe and effective user interactions. From the perspective of pure, logical investigations, perhaps the interface is not so important. Yet, it must play a prominent role in any practical, *interactive* defeasible reasoning system, including Decision Support Systems. The EVID program appears to embody the first serious attempt to characterize and implement a suitable logical interface for such systems.

Another feature of EVID that seems to be unique is the requirement that all application predicates be typed as either definite or defeasible, and that definite and defeasible rules have syntactic forms associated with these typings. This "strong typing" is based on the epistemological roles of the predicates, not on features that would typically be used for data typing in a programming language. EVID's typing requirements impose constraints on the form of knowledge representation used in an EVID-style application program. Yet, I do not believe that these constraints are oppressive, and it could be argued that they encourage the "knowledge engineer" to practice careful thinking in the design of knowledge based systems. The typing requirements certainly require one to consider carefully which kinds of KB facts should be treated as "epistemologically basic" and which should be "defeasibly inferred". Yet, I suspect that EVID typings will be considered unimportant by some researchers; it would be an interesting project to attempt to emulate EVID's functionality without these typings.

If one reviews the literature on nonmonotonic reasoning, e.g., in [4], one finds many attempts to develop theoretical systems of nonmonotonic logic, which are usually *extensions* of the first-order predicate calculus. Aside from the internal problems with such systems, they are very difficult (if not impossible) to implement with any satisfactory degree of efficiency. Indeed, Ginsburg writes ([4], p. 11), "... implementations of nonmonotonic reasoning systems (and there are very few) tend to be excruciatingly slow because of the repeated need for consistency checking." The approach taken with EVID is modest; rather than attempt to implement an entire first-order system, EVID merely interprets and interacts with a normal Prolog program which uses some metapredicates. Of course, an EVID-based application program uses negation by failure, most often together with **defeated_for** and **defeated**, but Prolog is capable of handling such queries quite efficiently.

One of the best known systems of nonmonotonic reasoning is Reiter's default logic; see

[10] (reprinted in [4], to which the following page references apply). Reiter's system augments first-order predicate calculus with so-called "default rules" of inference, so it is an extension of a full standard logic. A default inference involves assuming a proposition "...in the absence of any information to the contrary..." (p. 68). There are several forms of default rules, but a very simple example is this (restated from the one Reiter has on p. 68): 'If *bird(X)* and it is *consistent* to assume that *flies(X)*, then infer *flies(X)*.' As Reiter points out (p. 69), it is difficult to provide a suitable formal characterization of this consistency requirement, and he devotes much effort to doing so. An obvious inconsistency can result from '$penguin(X) \rightarrow \neg flies(X)$', where $\neg$ is standard, first-order logic negation. Now, regardless of the formal characterization details, inconsistency in a Reiter system always seems to depend on the use of $\neg$, so any defeater must be absolute in an even stronger form than is given by EVID's **defeated** predicate. Furthermore, there does not seem to be any simple way of expressing relative defeats in a Reiter system. If a program has more than one default rule for **concl**, and one of these rules is defeated, then the inference is blocked from all of them.[7] In my judgment Reiter's default logic suffers from a fundamental conceptual error of interpreting "absence of contrary information" in terms of logical consistency requirements on the sentences of a theory. It appears that defeasible reasoning cannot be adequately explicated in terms of simple consistency requirements, and the design of EVID avoids this particular error.

If the above is a correct interpretation of Reiter's default logic, then that system will be severely limited in its ability to provide convenient treatment of relative defeaters. The simple example program using **ucw** should demonstrate the importance of representing multiple kinds of evidential conditions and relative defeaters for these. Another system that has similar difficulties is "d-Prolog", described in [6]. Since its syntax is somewhat different from that used by EVID, it is not immediately clear how best to formulate the **ucw** example in d-Prolog. Yet, there is no *direct* way to express relative defeater rules in it, and trying to follow the standard d-Prolog program styles always leads to rule systems which contain only absolute defeaters. Hence, if **ucw(bob)** follows defeasibly from two sets of evidence, and only one of them is defeated, then the program behaves as if they both were.

---

[7] As mentioned in Section 1, J. Pollock has previously distinguished "rebutting" from "undercutting" defeaters. In [7], p. 37, without going into details, he generally criticizes the A.I. literature on nonmonotonic reasoning for not making this distinction. Although Pollock's work is related to mine, I will not attempt a comparison, since I do not have enough information about the implementation details of Pollock's system. It does appear from [8] that his system (if implemented as described) would have a much more complex logic than an EVID system. On the other hand, it does not appear that he is particularly concerned with an interactive user interface.

One system that appears capable of handling relative defeats is described in [2]. Brewka's approach uses some aspects of Reiter's work and also of some earlier modal logic nonmonotonic systems. The basic idea is to state default rules and to *name* these rules, e.g., r_1, r_2, etc. The rules are formulated as sentences, rather than as rules of inference. Brewka introduces a metapredicate, *appl*, such that *appl*(r_1,*X*) means that the rule r_1 applies to the object *X*. (Actually, there must be several such predicates of different arities corresponding to the number of free variables in the rule.) Finally, in addition to default rules (stated as sentences), Brewka uses ''exceptions'' such as '*penguin*(*X*) $\rightarrow \neg$ *appl*(r_1,*X*)', where r_1 might be the default rule that says that birds fly. This work was motivated by a concern for handling interacting defaults, and Brewka does not explicitly discuss evidential support and blocks of such support. Instead, his approach is to block the application of a rule. However, it would appear that this system could handle relative defeats.[8] A recently developed system that also ''names'' default rules with special predicates is described in [9]. This program (like Brewka's) allows explicit blocking of an application of a default rule by a (named) reference to that rule. The manner of naming rules and its use in deductive procedures is unusual, but does yield considerable flexibility. It is somewhat difficult to see a natural, intuitive interpretation for the special predicates that are introduced to name or refer to the default rules.

Many other comparisons can be made, but most of them pertain to somewhat specific problems and issues. Also, I avoid any detailed comparisons with Circumscription approaches. At present there are a number of different theoretical approaches to Circumscription, see [4] and [3], but few implementations. It is difficult to know where to begin a systematic comparison. About all I can say is that the **defeated_for** and **defeated** predicates were introduced after some earlier experiments with an **abnormal** predicate that was similar to those used in Circumscription theories. In my opinion, the use of an **abnormal** predicate, which is common in the Circumscription literature, seems misleading, *ad hoc*, and awkward.

To be fair, it should be pointed out that a prime motivation of many investigators has been to develop systems that automatically infer certain kinds of defeasible conclusions in situations where it would appear that the KB has conflicting conclusions. The conflicts often arise because of the use of $\neg$. For example, suppose that we have the rules: *'flies(X) if(defeasibly) bird(X)'*, *'$\neg$ flies(X) if(defeasibly) penguin(X)'*, and *'bird(X) if penguin(X)'*. If *penguin(peggy)*, what

[8] I have left out some details. Brewka's system also uses a modal operator corresponding to ''is consistent'', similar to that used in Reiter's system. The overall design is rather complex.

should be inferred about her flying ability? One reasonable approach, which has a long history in scientific methodology, is to use the most specific information available, namely, that *peggy* is a penguin, and to infer that she does not fly. The d-Prolog system of [6] is one that uses such an approach. Internally, the program compares degrees of specificity of predicates in order to decide which of the competing rules to use.

EVID could perhaps be augmented with specificity comparison mechanisms, and they might be useful in some applications. Fortunately, however, such comparisons will often not be required because they will be an automatic result of EVID-style knowledge representations. To see this, note that the above example would be represented in EVID by the following rules:

```
flies(X) :-
    bird(X),
    not( defeated_for( bird(X), flies(X) ) ).

defeated_for( bird(X), flies(X) ) :-
    bird(X),
    penguin(X).

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

In these rules, **bird** and **penguin** are definite, and **flies** is defeasible. Although this representation does not *mean* precisely the same as the preceding, it automatically keeps penguins from flying, and it is useful for applications. Moreover, in EVID if **penguin(peggy)**, then **whynot(flies(peggy))** explains that the evidential support **bird(peggy)** is relatively defeated by **penguin(peggy)**.[9]

[9]On the other hand, d-Prolog is able to make some additional modality distinctions that are not built-in features of EVID. This is partly because d-Prolog uses a genuine negation, whereas EVID's design is limited to negation by failure. Yet, by including additional metapredicates in an application program, one can often simulate special modal distinctions that might be desired for particular applications.

6. Intended Applications

6.1. Inheritance Hierarchies. The literature on nonmonotonic reasoning contains a number of standard test cases for reasoning systems. These cases consist of small puzzle-like problems to check whether a program draws the allegedly correct defeasible inferences from specified assumptions. EVID-style programs have been written for all of the major types of standard test cases. When these examples are translated into appropriate representations for EVID, the behavior of the programs is very reasonable. These little programs will not be described here because it is more interesting to consider some possible practical applications.

Another kind of standard test is to build a KB which has a hierarchical arrangement of many "classes" of things, say, species of animals. Subclasses are to inherit properties of their superclasses, but this inheritance is usually defeasible, in the sense that an inherited property can be overridden by a more specific property of the subclass. For instance, most kinds of birds will inherit "can fly" from the general class of birds, but this inheritance will be blocked for penguins, ostriches, and some others. A hierarchy of this kind may, in addition to classes, have nodes that represent individuals, such as Leo, a particular lion. Finally, it is possible that a given class or individual may have more than one superclass, giving rise to the possibility of a conflict over which superclass should transmit the value of a particular inheritable property. This is sometimes called "the problem of multiple inheritance."

An inheritance hierarchy with all of the above features has been implemented as an EVID application program. The basic approach is straightforward, although the details can be tedious to program. Inheritance from a superclass is handled by letting membership in the superclass serve as evidence (usually defeasible) for the inherited property. This representation requires no special treatment to obtain multiple inheritance, one merely writes two or more such inheritance rules. Inheritance conflicts are obvious because a query for the value of the inherited property will yield more than one answer. It is then the responsibility of the knowledge engineer to write suitable **defeated_for** rules to block the inheritance of the inappropriate answers. This form of representation can lead to many rules, so programs of this type will not be as efficient as those using a directly procedural approach (such as semantic nets and frame systems). On the other hand, an inheritance hierarchy interpreted by EVID offers many additional features, including explanations and explicit user overrides by means of **addit**, **defeatit_for**, etc. Also, special forms of rules permit the knowledge engineer to represent different ways in which an individual

may be an exception to a default rule. Overall, EVID programs offer extensive representational flexibility.

Inheritance hierarchies are interesting, but I do not consider them, *per se*, to be application programs. Rather, they are a convenient knowledge representation for possible applications. Suffice it to say that EVID can handle such hierarchical representations when needed, although large programs may (as always) run into computational efficiency problems. In the remainder of this section, I will briefly describe two small experimental application programs that have been successfully run in the EVID environment. These examples are selected because they demonstrate two general types of applications for which EVID may be especially useful. The first type can very roughly be described as *user directed* because the user specifies the kinds of conclusions that interest him, and he enters data appropriate to obtaining these conclusions or at least some inferences relevant to them. The second type is *program directed* because the program is designed to seek a certain kind of conclusion goal and it queries the user for information leading to this goal. Well known expert system programs for classification and diagnosis fall into this second category.

6.2. A User Directed Application. The first example application is a fictional military intelligence adviser. EVID was originally designed specifically in order to implement this kind of intelligence adviser, which in turn is a model for many types of DSS's. Suppose there is a primitive mountainous island and a spy is expected to travel from point A on one side to point B on the other side. The application program's KB comprises many rules, both definite and defeasible, about possible modes of travel from A to B depending on the traveler, weather conditions, and other factors. To use the program, we enter various current facts, and then it advises us about how the spy is expected to travel. Naturally, this advice is based on defeasible inferences that the program makes and these inferences may change with new information. This program continues to grow in size as new and more complex rules are added to it. I will only give an informal description of some of its features, but will try to convey an accurate impression of its functionality.

There are only three general ways the spy can travel from A to B; these are specified by **travels_by(Person,X,Date)**, where **X** can take the values **land, air**, or **sea**. The only way to travel by land is to take one of three roads: **highroad, lowroad**, or **midroad**, expressed by **takes(Person,X,Date)** in which **X** is the road. The only way he can travel by air is to fly something, specified by: **flies(Person,X,Date)**, where **X** can have many values such as

single_prop, jet, balloon, etc. Finally, it is assumed that the only way he can travel by sea is if he rides a boat, **rides(Person,boat,Date)**. These four predicates are the only defeasible predicates in the program, which also has about a dozen definite predicates pertaining to such things as weather conditions, what types of air craft the spy can fly, whether or not he has tire chains, is acrophobic, gets seasick, etc. There is also a predicate that specifies that he prefers some modes of travel over others, and a predicate that specifies that certain information about him is uncertain (i.e., based on questionable intelligence sources). The program currently has 24 rules. Of these, two are definite, and nine are defeasible rules, while thirteen are defeaters (including three contra-defeaters).

We begin a dialogue with the program by telling it, say, that **travels(spy,oct31)**, where **spy** is a generic name for an unidentified spy. If this is all that it is given (other than the fact that **spy** is a person), then it defeasibly infers that he takes the highroad. If we then tell it that there is mountain snow on October 31, it infers that he takes the lowroad. But if we also tell it that he has tire chains, then he takes the highroad in spite of snow. There are other conditions that defeat his taking the highroad. Suppose that he is on the lowroad, and we give the program new information that there is a coastal storm, then he takes the midroad. At any point, we can use **howgetit** and **howdefeatit** to help us decide what additional, external information would be relevant to different conclusions. Also, **why** and **whynot** yield the program's reasons for currently inferring or not inferring certain conclusions.

Now suppose that the user is an experienced intelligence officer. Such a person may notice a special nuance of the situation that is not covered by the program's KB. For example, this could be some peculiarity of the sea's condition, or some particular information about who the spy is and what his dispositions are, etc. A detailed, reliable KB is very difficult to construct, and an experienced human expert, working in a real world context, will almost always be able to draw defeasible inferences, and make defeats of such inferences, in ways that go beyond the capabilities of an automated system. The automated system can be very helpful, but it should only be treated as a smart, friendly adviser. EVID gives the user the ''last word'' subject to the constraints of its logical interface. Thus, if the program has defeated a certain conclusion, the user can override this and add that fact back into the system. If the program has drawn a conclusion, the user can defeat it, again subject to the constraints of EVID's logical interface. A wise user will only perform such actions with great care. Whether he ultimately agrees with the program, or overrides it in some respects, he will have responsibility for the decisions that are

made.

To take a specific example, suppose that the user has information that the spy in question is **bob** and this user knows that **bob** prefers flying over taking any road. Also, suppose that there is mountain snow, and that bob can fly a single prop plane. When all this information is entered, one might expect the program to infer that bob flies, but this is not the case because of a rule that has mountain snow defeating the flying of a single prop, so **bob** rides a boat and travels by sea. If we subsequently get and enter new information that **bob** has had new training and can now fly a jet, then the program infers that he flies it and that he travels by air. Finally, suppose that the user also knows that **bob** has reason to believe that the user knows about **bob**'s new training, so **bob** is likely to expect anti-air travel preparations from the user. Thus, the user concludes that, at least in these special conditions that go beyond the program's KB, **bob** will avoid air travel after all. Then the user can use **defeatit** to overrule the air travel, and the program will conclude that **bob** rides a boat.

The current implementation of this test program uses a standard command line interface, but this program is part of a larger project to investigate possible designs for *logical spreadsheets*. Several possibilities suggest themselves, but here is one sketch that easily comes to mind. In a typical application, there will be a set of data that is externally obtained and supplied to the program. This data will usually be expressed in terms of definite predicates, such as the ones mentioned above. There will also be other facts that the program infers, either positively or defeasibly, from the input data and its KB. One could set up a kind of tabular spreadsheet, with cells (or windows) for the input data, and other cells for displaying the output conclusions. Some user actions, like **defeatit**, could be performed with the help of menu selections. As new information is fed into the spreadsheet, it would be updated to show how the program's inferences change. A separate window would be useful for display of the program's ''explanations'' (responses to **why**, etc.). A spreadsheet of this sort would be convenient for answering logical ''whatif'' questions.

6.3. A Program Directed Application. There is no standardized definition for ''expert system,'' but typical examples of such systems help to solve diagnosis and classification problems. In fact, many diagnostic expert systems do little more than classification: they have a KB that includes a classification of different types of functional abnormalities (diseases, breakdowns, poor performance, etc.). The classification system is largely based on observational data (symptoms, laboratory test results, case histories, etc.). The expert system includes a large

number of heuristic rules that lead from entered observational data towards the formulation of classification hypotheses. Because of the classificational nature of such programs, it is common to write experimental prototypes for the identification of an animal specimen's species in terms of the observed data about the specimen. Such a program typically includes an interface that prompts the user to enter certain observational data in response to the program's questioning. This data then fires internal rules that lead to partial identifications and to additional questioning. When successful, this process eventually leads to a few (often one) identificatory hypotheses which are sometimes associated with probabilities or certainty factors.

One of the commonly heard complaints about such programs is that they are not sufficiently flexible to satisfy a well-informed user. For instance, often the user can justifiably exclude many possibilities himself, but the program will nevertheless question him with many seemingly irrelevant questions pertaining to these possibilities. In other words, many expert systems follow a fairly standard pattern of questioning and do not let the user "guide" the system more quickly towards the desired goal. Also, typical expert systems do not use defeasible reasoning, at least not in a fashion that is guided by an overall conception of what the role of such reasoning should be in an expert system. It would seem that effective use of defeasible reasoning could lead to more flexible and efficient expert systems.

EVID was not originally designed for building expert systems, but I have been experimenting with a simple animal classification system that runs in EVID. There are two general motivations for these experiments: (i) to study how defeasible rules can be used in the system to increase its flexibility and efficiency, and (ii) to give the user the full benefits of EVID's logical interface, in particular, so that he will be able to use **addit** to enter information directly and then avoid having the program ask him questions pertaining to such entered information. My experimental expert system has four main parts:

- The EVID shell (used without any modifications)
- The expert system's control mechanisms for querying the user
- The defeasible domain rules for generating classificatory hypotheses
- Other domain rules for making internal defeasible inferences

When achieved, the final identifications are in terms of specific animal kinds such as tiger, bald eagle, platypus, etc. Intermediate classifications include larger categories such as mammal, bird, carnivore, etc. The rules for generating hypotheses are fairly standard in one way – they

acceptable speeds in both systems. For portability, the present code is almost entirely compatible with the *de facto* standard Edinburgh Prolog. Developing a logical spreadsheet, as described in the previous section, would presently require much extra, system-dependent code.

References

[1] M. Belzer and B. Loewer. A Conditional Logic for Defeasible Beliefs. *Decision Support Systems* 4 (No. 1):129-142, 1988.

[2] G. Brewka. Tweety – Still Flying: Some Remarks on Abnormal Birds, Applicable Rules, and a Default Prover. In *Proceedings of AAAI-86, Vol. I*, pages 8-12. Morgan Kaufmann, San Mateo, California, 1986; ISBN: 0-934613-13-3.

[3] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, California, 1987; ISBN: 0-934613-31-1.

[4] M. L. Ginsburg (editor). *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann, San Mateo, California, 1987; ISBN: 0-934613-45-1.

[5] S. O. Kimbrough and F. Adams. Why Nonmonotonic Logic? *Decision Support Systems* 4 (No. 1):111-127, 1988.

[6] D. Nute. Defeasible Reasoning and Decision Support Systems. *Decision Support Systems* 4 (No. 1):97-110, 1988.

[7] J. L. Pollock. *Contemporary Theories of Knowledge*. Rowman & Littlefield, Totowa, New Jersey, 1987; ISBN: 0-8476-7452-5.

[8] J. L. Pollock. Defeasible Reasoning. *Cognitive Science* 11 (No. 4):481-518, 1987.

[9] D. Poole. A Logical Framework for Default Reasoning. *Artificial Intelligence* 36 (No. 1):27-47, 1988.

[10] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence* 13 (No. 1-2):81-132, 1988.

[11] L. Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge & Kegan Paul, London, 1922.