Chapter 6

Proteus: A Frame-based Nonmonotonic Inference System

David M. Russinoff

Introduction

Early artificial intelligence systems relied on first-order predicate logic as a language for representing domain knowledge. While this scheme is completely general and semantically clear, it has been found to be inadequate for organizing large knowledge bases and encoding complex objects. As an alternative, various frame-based languages have been employed. These languages are designed to support the natural representation of structured objects and taxonomies. They have proved to be well-suited for representing many useful relations, although they lack the general expressive power of the predicate calculus.

Knowledge-based systems may also be classified according to inference methods. Most deductive systems may be characterized as either goal-directed (backward chaining) or data-directed (forward chaining). In a goal-directed system, logical implications are encoded as rules that are used by the system to reduce goals to simpler subgoals. This allows knowledge to be represented implicitly, without using space in the knowledge base, until it becomes relevant to a current problem. In this framework, however, it is difficult for the knowledge base designer to build control into a system. Data-directed inference, on the other hand, is based on production rules, which the system uses to derive all logical consequences of new data automatically. While control of inference is more natural within this paradigm, it uses space less efficiently, representing all knowledge explicitly.

This chapter describes the knowledge representation and reasoning components of *Proteus* [Russinoff 1985b, Petrie 1987, Poltrock, et al. 1986], a hybrid expert system tool, written in Common Lisp, under development at the Microelectronics and Computer Technology Corporation. Proteus is frame-based, but allows knowledge to be expressed in terms of arbitrary predicates. It also integrates goal-directed and data-directed inference, allowing the knowledge engineer the freedom to decide whether each logical implication is more suitably represented as a backward rule or a forward rule.

A central feature of Proteus is a nonmonotonic truth maintenance system (TMS), based on [Doyle 1979], which records logical inferences and dependencies among data. This allows efficient revision of a set of beliefs to accommodate new information, the retraction of a premise, or the discovery of a contradiction [Petrie 1987]. It also facilitates the generation of coherent explanations. Data dependencies and truth maintenance in Proteus are discussed in Section 2.

In Section 3, we introduce frames, along with classes, attributes, metaclasses, and types. Section 4 describes simple data, including attribute values that are attached to frames, as well as assertions associated with predicates. Here we discuss the use of the TMS in connection with single-valued slots and inheritance.

Finally, we describe the Proteus inference system and its integration with the TMS. Sections 5 and 6 deal with backward and forward chaining, respectively.

1 Truth Maintenance

Each element of the Proteus database represents a potential belief. The status of this belief, which is subject to change, is reflected in the *support-status* of the datum, the value of which may be `IN`, indicating that it is currently believed, or `OUT`, indicating current disbelief. This value is assigned by the TMS in accordance with a list of *justifications* that have been attached to the datum.

Each justification consists of a pair of lists of data, the `IN`-list and the `OUT`-list of the justification. A justification is said to be *valid* and is considered to represent reason for belief in its associated datum if each element of its `IN`-list is `IN` and each element of its `OUT`-list is `OUT`. The justified datum is said to depend *monotonically* on each member of the justification's `IN`-list and *nonmonotonically* on each member of the `OUT`-list.

Also associated with each datum is a list of other data called its *supporters*. The supporters of a datum are considered to be responsible for its current support-status.

It is the function of the TMS to assign support-statuses and supporters to data in a manner that is consistent with their justifications, and to adjust these assignments continually as required by the addition of new justifications and the retraction of old ones. More precisely, the state of the database, as constructed by the TMS, must satisfy two requirements: *stability* and *well-foundedness*. A *stable* state is one that satifies the following conditions:

1. A datum is `IN` if it has at least one valid justification. In this case its list of supporters is the result of appending the `IN`-list and `OUT`-list of one of its valid justifications. This justification is identified as the *supporting justification*.

2. A datum is `OUT` if it has no valid justification. Its supporters then include one representative of each of its invalid justifications: either an `OUT` member of the `IN`-list or an `IN` member of the `OUT`-list.

The requirement of well-foundedness is that no set of beliefs be mutually dependent, i.e., there may be no sequence of data $d_0, \ldots, d_n$, all of which are `IN`, such that $d_0 = d_n$ and for $i = 1, \ldots, n$, d_{i-1} is a supporter of d_i.

An example of an admissible state is shown in Fig. 1. In this graph and

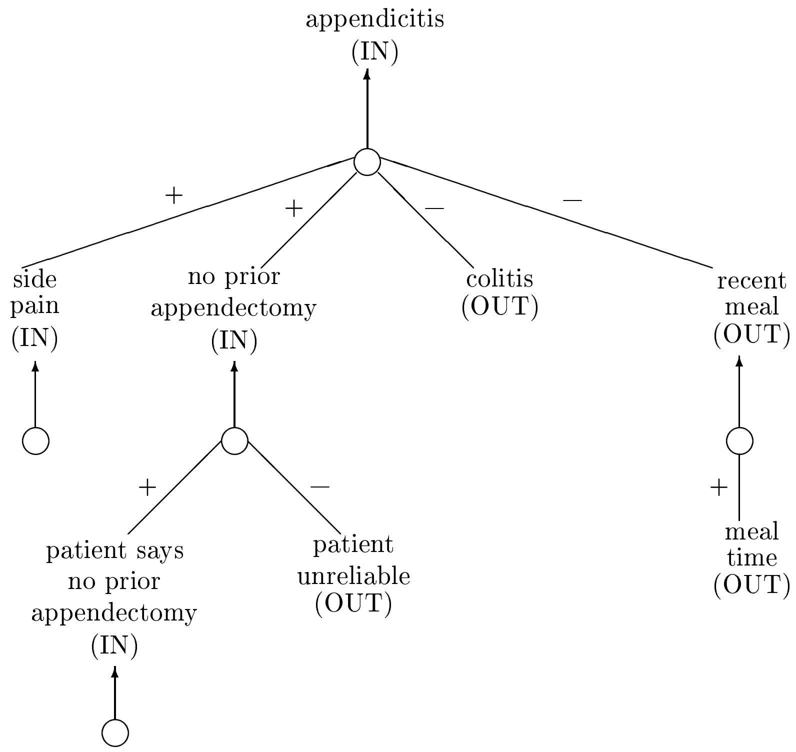

Figure 1: A Stable Well-founded State

those that follow, each circle corresponds to a justification, with an arrow pointing to the justified datum, positive arcs connected to the elements of the `IN`-list, and negative arcs to elements of the `OUT`-list. Thus, the datum representing a diagnosis of appendicitis has a valid justification with a two-element `IN`-list and a two-element `OUT`-list. The belief that the patient has a side pain is supported by a justification with an empty `IN`-list and an empty `OUT`-list and is said to be a *premise.* The datum representing the unreliability of the patient has an empty list of justifications and is therefore `OUT`. If this datum were to acquire a new valid justification, then its support-status as well as those of the data that depend on it (directly or indirectly) must be reevaluated, ultimately forcing the diagnosis `OUT`. This phenomenon, the

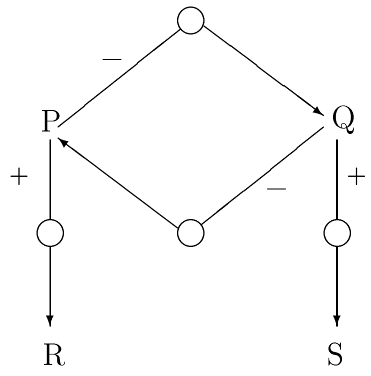

Figure 2: Alternative Assumptions

development of a new belief resulting in the abandonment of an old one, characterizes *nonmonotonic reasoning*.

In the presence of nonmonotonic dependencies, the status-assignment problem may not have a unique solution. In the situation shown in Fig. 2, the TMS may succeed either by making P (and hence R) `IN` and Q (and hence S) `OUT`, or by giving the opposite assignments. This choice between alternative hypothetical assumptions can only be made arbitrarily, and may have to be revised later as new justifications are produced (e.g., if Q acquires a new valid justification while P is `IN`).

Circularities involving nonmonotonic dependencies may also impose unsatisfiable constraints on the TMS, a situation that may be difficult to detect. Two simple examples of this are shown in Figs. 3 and 4. Note that the network of Fig. 4 does have a stable state (in which all data are `IN`), but this state is ill-founded, and therefore inadmissable.

As described in [Russinoff 1985a], the Proteus TMS is *complete* in the sense that given any database with any set of justifications, it will achieve a

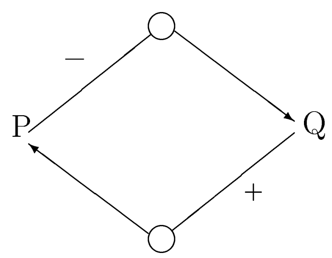

Figure 3: Unsatisfiable Dependencies

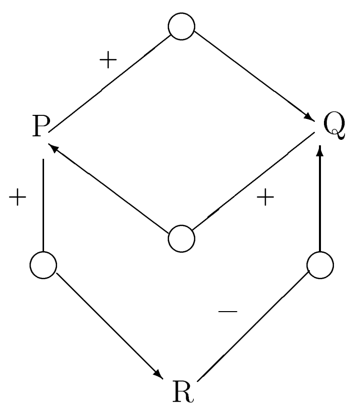

Figure 4: No Well-founded Stable State

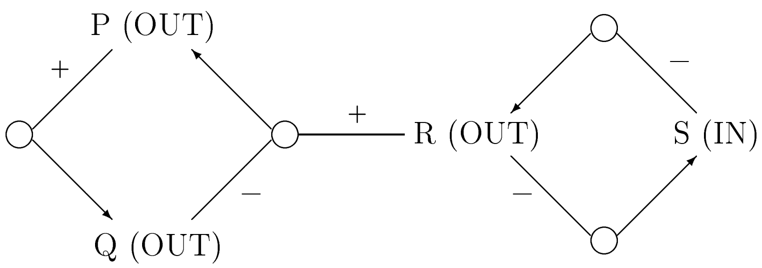

Figure 5: A Solvable Odd Loop

stable well-founded state if such a state exists, and otherwise will recognize and report failure. This represents an improvement over the original TMS of Doyle [Doyle 1979], as well as other published procedures for truth maintenance [Charniak, et al. 1980, Goodwin 1986]. These systems all fail (perhaps even fail to terminate) in the presence of certain circular dependencies that have been characterized as *odd loops.* An odd loop is a cycle of arcs with an odd number of minus signs, as in Figs. 3, 4, and 5. A dependency network containing such a loop may or may not be satisfiable. (The network in Fig. 5 does admit a solution.) While the presence of odd loops complicates the truth maintenance task and is generally considered undesirable, they are sometimes unavoidable in practice, particularly in dependency networks that are based on input from several users.

In the sequel, we show how data dependency networks are created in Proteus by the user, by the frame system, and by both forward and backward inference.

2 Frames and Classes

The data on which the TMS operates represent statements about objects. Before discussing the structure of these data, we shall describe the objects that they concern. These objects, called *frames*, are the subject of this section.

Figure 6: Built-in classes

2.1 Classes, Subclasses, and Members

In the initial state of the system, there exist several frames. One of these, named **CLASS**, plays a special role as discussed below. The others are **LIST**, **CONS**, **NULL**, **SYMBOL**, **NUMBER**, **FIXNUM**, and **SINGLE-FLOAT**. The user may enlarge this set by creating new frames, one at a time.

There are two primitive relations defined on frames: *instance* and *child*. If a pair (x, y) is an element of the instance relation, we say that *x is an instance of y*, or that *y is the type of x*. For a pair (x, y) in the child relation, we say that *x is a child of y*, that *y is a parent of x*, or that *x is linked to y*. Fig. 6 depicts these relations as they are defined in the initial state. Broken lines are drawn from instances to types; solid lines from children to parents. Thus, **CLASS** is the type of every system-defined frame.

Two other important relations are defined in terms of these primitives:

subclass and *member*. The subclass relation is defined as the reflexive transitive closure of the child relation. Thus, x is a subclass of y (equivalently, y is a *superclass* of x) if either x is identical to y, or x is a child of a subclass of y. The membership relation is defined as follows: x is a member of y if x is an instance of a subclass of y. In this case we may also say that x *is a* y. In particular, a *class* is by definition a frame that is a member of **CLASS**. Note that every system-defined frame is a class, including **CLASS** itself.

As new frames are created by the user, he may also extend these relations by assigning types to instances and creating links (between user-defined classes only). This must be done in such a way, however, that at each stage of the development, the following properties are preserved:

1. The instance relation is a function, i.e., for each frame x there exists a unique frame y such that x is an instance of y.

2. **CLASS** is the only frame that is an instance of itself. Thus, whenever a new frame is created, some preexisting frame must be specified as its type.

3. The subclass relation is a partial order. That is, if x is a subclass of y and y is a subclass of x, then x and y are identical.

4. If x is an instance of y, then y must be a member of **CLASS**.

5. If x is a child of y, then x and y must both be members of **CLASS**.

Thus, according to the last two of these properties, only classes may have instances, members, children, parents, subclasses, or superclasses. There is a further restriction on the classes that may be instantiated by the user: a user-defined frame may be an instance of **CLASS** or of any user-defined class, but it may not be an instance of **LIST**, **CONS**, **NULL**, **SYMBOL**, **NUMBER**, **FIXNUM**, or **SINGLE-FLOAT**. Instead, whenever the system encounters a Common Lisp object whose datatype is the name of one of these system-defined classes, the object automatically becomes an instance of the named class. For example, if the number *3* is read, it becomes an instance of **FIXNUM** and thus a member of **NUMBER** (in other words, a number). When the symbol `NIL` is encountered, it is recognized as the unique instance of the class **NULL**, and hence both a list and a symbol.

An example of a user-defined system of frames is illustrated in Fig. 7. This example involves eight new classes, all of which are subclasses of the class **PERSON** and instances of the class **CLASS**. For clarity, classes are denoted in bold-face and other user-defined frames in italics. *SHIRLEY*, for example, as an instance of **TA**, is not a class, but is a TA, a graduate, a staff, a student, an employee, and a person.

2.2 Metaclasses

Of course, the existence of classes as frames provides the advantage of being able to reason about classes at the same level at which one reasons about the objects of which they are comprised. It is often desirable to be able to reason

Figure 7: User-defined Classes

about classes of classes as well. The only class we have seen so far that has classes as members is **CLASS** itself. Any class that has a class as an instance must be a subclass of **CLASS**, in which case all of its members are classes. Such a class is called a *metaclass*. While **CLASS** is the only system-defined metaclass (and the only one in the example of Fig. 7), additional metaclasses may be created simply by linking any user-defined classes to **CLASS**.

The example of Fig. 8 includes several user-defined metaclasses (denoted in large bold print). This example is based on six classes of animals: **ANIMALIA** (the class of all animals) and five of its subclasses, which are related as indicated by the solid arrows connecting them. These classes could be constructed simply as instances of **CLASS**. But in order to represent and utilize the knowledge that these classes share more than mere classhood, we first define a metaclass called **BIOLOGICAL-CLASS**, intended to include the animal classes among its members. In fact, the animal classes are partitioned into smaller metaclasses by defining them as instances of **KINGDOM**, **PHYLUM**, and **SPECIES**, which are subclasses of **BIOLOGICAL-CLASS**. Note that **KINGDOM**, **PHYLUM**, and **SPECIES** are themselves not simply instances of **CLASS**, but are classes by virtue of being instances of the metaclass **TAXONOMIC-DIVISION**. Moreover, they are metaclasses by virtue of their links to the metaclass **BIOLOGICAL-CLASS**.

Figure 8: User-defined Metaclasses

2.3 Attributes

Corresponding to each user-defined class is a (possibly empty) set of user-defined *attributes* associated with that class. Every attribute is defined for a unique class — the frames that may assume values for a given attribute are the members of the class for which it is defined. Thus, in the example of Fig. 7, if attributes called `HOURLY-WAGE` and `TITLE` are defined for the classes **EMPLOYEE** and **FACULTY**, respectively, then the frames that may have `HOURLY-WAGE` values are *NAT*, *SHIRLEY*, *AHMET*, and *DONALD*, while only *AHMET* and *DONALD* may assume `TITLE`s. These attribute values form a classification of data to be discussed in Section 4.1.

The manner in which attributes are inherited from their defining classes provides motivation for the construction of user-defined metaclasses, such as those of Fig. 8. If **ANIMALIA**, **CHORDATA**, etc. had simply been defined as instances of **CLASS**, there would have been no way for them to acquire attribute values. But as members of the user-defined class **BIOLOGICAL-CLASS**, they may assume values for any attributes defined for that class. Thus, if `COMMON-NAME` is among these attributes, then the statement "the `COMMON-NAME` of **PROTOZOA** is `ONE-CELLED-ANIMAL`" makes sense. The partitioning of **BIOLOGICAL-CLASS** into subclasses allows the definition of attributes that pertain to some biological-classes but not to all. For example, one might refer to the number of species of protozoa, but not the number of species of amoeba, while the number of species of animalia is not practically measurable. The attribute `NUMBER-OF-SPECIES`, therefore, should not be defined for the class **BIOLOGICAL-CLASS**, but rather for its subclass **PHYLUM**.

2.4 Variables and Types

Along with the objects that appear in Proteus data, there are also occurrences of variables. These are denoted as symbols with initial character "?". Variables, as usual, represent unspecified objects. The process of unification, which is central to the mechanisms of forward and backward chaining, is built on the basic operation of binding (i.e., assigning values to) variables. An unbound variable `?X` may be bound either to an object or to another unbound variable `?Y`. In the latter case, if `?Y` is subsequently bound, its binding also becomes the binding of `?X`.

Since objects in this system are classified by their types, it is natural and useful to classify variables in a similar way. Thus, following [Aït-kaci and Nasr 1985], a variable may be specified to be of a certain type. This is done by appending the name of a class to the variable name, using ":" as a separator, as in `?X:STUDENT`. The consequence of assigning a type to a variable is that any binding of the variable is required to be a member of the variable's type.

In the presence of typed variables, the standard unification algorithm must be altered in several ways. First, before a variable is bound to an object, it must be verified that the object is a member of the variables's type. Second, before a variable is bound to another variable, it must be verified that the types of the two variables are compatible, so that it will be possible later to bind them to the same object. Thus, the two types must have a nontrivial common subtype. Finally, when a variable `?X` is bound to a variable `?Y`, the type of `?Y` must be replaced in order to ensure that any later binding of `?Y` is consistent with the type of `?X`. The new type of `?Y` should be the most general common subtype, or *greatest lower bound*, of the type of `?X` and the old type of `?Y`. For example (see Fig. 7), a variable `?X:UNDERGRADUATE` could be bound to a variable `?Y:STAFF`, with type of `?Y` replaced by **GRADER**. `?Y` could then be bound to *NAT*, but no longer to *SHIRLEY*, which would violate the type restriction on `?X`.

Thus, the modified unification algorithm depends on the computability of the greatest lower bound (g.l.b.) of two variable types. Unfortunately, the partially ordered set of classes does not form a *lattice*, i.e., the g.l.b. of two classes may not exist. The classes **STUDENT** and **EMPLOYEE** of Fig. 7, for example, have two common subclasses, **GRADER** and **TA**. Since neither of these is a superclass of the other, neither can be said to be the g.l.b. of **STUDENT** and **EMPLOYEE**.

This problem is solved by generalizing the notion of *type*. A type is now defined to be a set of classes, none of which is a subclass of another. A type t_1 is a subtype of a type t_2 if each of the classes of t_1 is a subclass of at least one of the classes of t_2. An object is said to belong to a type if it is a member of at least one of its classes. It follows that an object belongs to a type if it belongs to any of its subtypes.

Under this ordering, the set of all types forms a lattice. The type of a variable may be any member of this lattice. The identification of each class c with the type $\{c\}$ induces an embedding of the partially ordered set of classes into the lattice of types. In this context, the g.l.b. of two classes may always be computed. Thus, the g.l.b. of **STUDENT** and **EMPLOYEE** is the type $\{$**GRADER**,**TA**$\}$.

The empty set of classes, denoted **bottom**, is a subtype of every type. This type contains no objects and is not allowed as the type of a variable. If the g.l.b. of the types of two variables (e.g., `?X:STAFF` and `?Y:FACULTY`) is *bottom*, then these variables cannot be unified; their types contain no common objects.

The set of all maximal classes is also a type, denoted **top**. It is a supertype of every type, and every object belongs to it. If no type is specified for a variable, then the variable's type is taken to be *top* as a default. There is no restriction on the binding of such a variable.

The cost of the expressive power of typed variables is the resulting complication of the unification algorithm. In order to minimize this cost, the g.l.b. operation must be a fast computation. This is accomplished by means of an encoding scheme that associates with each class c a bit-string $\mathcal{B}(c)$, in such a way that c_1 is a subclass of c_2 if and only if $\mathcal{B}(c_1)$ is (bit-wise) less than or equal to $\mathcal{B}(c_2)$. For a type $t = \{c_1, \ldots, c_k\}$, $\mathcal{B}(t)$ is constructed as the logical-or of the $\mathcal{B}(c_i)$. The g.l.b. operation then reduces to logical-and. The details of this scheme are described in [Aït-kaci, et al. 1985].

3 Assertions

There are two main classifications of data: *assertions* and *rules*. Assertions, as described in this section, represent simple statements. Rules, which are used by the system to derive assertions from other assertions, are further classified as *forward rules* and *backward rules*. These are the subjects of Sections 5 and 6.

3.1 Instance Slot Values

Recall that a frame may assume values for a given attribute if it is a member of the class for which the attribute is defined. In this case, an *instance slot* corresponding to the attribute is attached to the frame. One or more values may be stored in this instance slot. Thus, if an attribute `DESCRIPTION` is defined for the class **PERSON** of Fig. 7, then the frame *DONALD* may acquire `DESCRIPTION` values. An assertion such as

```
@assert (description donald tall)
```

results in the creation of a datum called an *instance slot value*, which is stored in `DONALD`'s `DESCRIPTION` slot. This datum is justified it as a premise (i.e., with empty `IN`-list and `OUT`-list) and printed as

```
Instance Slot Value DESCRIPTION-1    (IN)
  (DESCRIPTION DONALD TALL)
```

When an attribute is initially defined, it is specified as either *single-valued* or *multiple-valued*. A frame may assume any number of coexisting values for a multiple-valued attribute. If `DESCRIPTION`, for example, is multiple-valued, then asserting a new `DESCRIPTION` for *DONALD* via

```
@assert (description donald young)
```

produces a new slot value, but has no effect on the old value. Thus, a query for *DONALD*s `DESCRIPTION` produces both values:

```
@?? (description donald ?X)

(DESCRIPTION DONALD TALL)
(DESCRIPTION DONALD YOUNG)
```

For a single-valued attribute, on the other hand, a frame may have only one effective value at any time. Suppose the attribute `NATIONALITY` is defined for class **PERSON** and declared to be single-valued, and that a `NATIONALITY` is asserted for NAT:

```
@assert (nationality nat french)

Instance Slot Value NATIONALITY-1    (IN)
  (NATIONALITY NAT FRENCH)
```

If another value is later asserted for the `NATIONALITY` of NAT, then the old value is overridden by the new one:

```
@assert (nationality nat swiss)

Instance Slot Value NATIONALITY-2    (IN)
  (NATIONALITY NAT SWISS)

@?? (nationality nat ?X)
```

```
(NATIONALITY NAT SWISS)
```

Actually, this restriction to a single value is enforced by the TMS: whenever a value is asserted for a single-valued attribute, it is added to the `OUT`-list of each justification of any preexisting conflicting value. Thus, the old value remains in the database, but is ignored in answering the query because it is now `OUT`. This method not only provides for the construction of explanations, such as

```
@why NATIONALITY-1

Instance Slot Value NATIONALITY-1    (OUT)
  (NATIONALITY NAT FRENCH)

was replaced by

Instance Slot Value NATIONALITY-2    (IN)
  (NATIONALITY NAT SWISS)
```

but also allows an old value to be reinstated if the overriding value is removed:

```
@erase (nationality nat swiss)

@?? (nationality nat ?X)

(NATIONALITY NAT FRENCH)
```

In fact, it always ensures that the value that is `IN` is the one with the most recently created justification that is currently valid. Note that if several conflicting values for a slot are asserted in succession, then a reassertion of the original value will result in a complicated dependency network, including odd loops. This implementation, therefore, requires a complete TMS as discussed in Section 2.

3.2 Class Slot Values

Values for an attribute may be attached to subclasses of its defining class as well as to its members. Each of these subclasses contains a *class slot*, the values in which may be inherited by any member of the class to which it belongs. For example, the command

```
@assert (description ?X:person aerobic)
```

adds a value to the `DESCRIPTION` class slot of **PERSON**:

```
Class Slot Value DESCRIPTION-2    (IN)
  (DESCRIPTION ?X:PERSON AEROBIC)
```

This datum represents the belief that every member of **PERSON** is `AEROBIC`. Similarly, any subclass of **PERSON** may acquire class values for this attribute:

```
@assert (description ?X:faculty pompous)

Class Slot Value DESCRIPTION-3    (IN)
  (DESCRIPTION ?X:FACULTY POMPOUS)
```

The `DESCRIPTION` values associated with a member of **PERSON** are then the instance values assigned to it specifically, along with the class values assigned to the superclasses of its type. Hence,

```
@?? (description donald ?X)

(DESCRIPTION DONALD TALL)
(DESCRIPTION DONALD YOUNG)
```

```
(DESCRIPTION DONALD POMPOUS)
(DESCRIPTION DONALD AEROBIC)
```

For single-valued attributes, a more complicated mode of inheritance is used. A value for a single-valued class slot may be inherited by a member of the class only if no value has been assigned to that member's instance slot. Class slot values for these attributes are therefore called *default values*.

Default values assigned to a given class override each other in the same manner as instance values for a given frame, so that only one default value assigned to a class may be `IN` at any time. If an attribute value is sought for a given frame, the frame's instance slot is first examined for an `IN` value. If there is none, then each of the superclasses of the frame is examined (in depth first order) until an `IN` default value is found.

Suppose that for the single-valued attribute `NATIONALITY`, the default values `AMERICAN`, `CHINESE`, and `INDIAN` are asserted for the classes **PERSON**, **STUDENT**, and **GRADUATE**, respectively. Suppose further that *NAT*'s `NATIONALITY` is asserted to be `FRENCH` and that *DONALD*'s is `GERMAN`. Then a query for the `NATIONALITY` values for all members of **PERSON** would produce the following:

```
?? (nationality ?X:person ?Y)
```

```
(NATIONALITY DONALD GERMAN)
(NATIONALITY AHMET AMERICAN)
(NATIONALITY SHIRLEY INDIAN)
(NATIONALITY NAT FRENCH)
(NATIONALITY DAVID CHINESE)
(NATIONALITY ESTELLE AMERICAN)
```

If a new default value were now asserted for **STUDENT**, only *DAVID*'s `NATIONALITY` would change:

```
@assert (nationality ?X:student texan)

?? (nationality ?X:student ?Y)

(NATIONALITY SHIRLEY INDIAN)
(NATIONALITY NAT FRENCH)
(NATIONALITY DAVID TEXAN)
```

If the default value for **GRADUATE** were retracted, then *SHIRLEY* would inherit from **STUDENT**:

```
@erase (nationality ?X:graduate indian)

?? (nationality ?X:student ?Y)

(NATIONALITY SHIRLEY TEXAN)
(NATIONALITY NAT FRENCH)
(NATIONALITY DAVID TEXAN)
```

3.3 Predicates and Assertions

Any attribute may be regarded as representing a binary relation whose domain is the set of members of some class. An instance slot value then corresponds to a pair of related objects, while a class slot value (at least for a multiple-valued attribute) corresponds to a set of such pairs. While this scheme is suitable for representing many kinds of information, it is somewhat restrictive.

One problem is that only relations of two arguments can be naturally represented in this way. Thus, the relation *x is in debt to y* may be realized as an attribute called `IN-DEBT-TO` defined for **PERSON**, and the statement *NAT is in debt to AHMET* is then asserted by assigning *AHMET* as a value to the frame *NAT*, i.e., by asserting `(IN-DEBT-TO` *NAT AHMET*`)`. On the other hand, the unary relation *x is in debt* could not be represented so naturally as an attribute. Of course, we could define a Boolean-valued attribute

`IN-DEBT` and represent *NAT is in debt* by asserting `(IN-DEBT` *NAT* `T)`, but it would be preferable to be able to assert `(IN-DEBT` *NAT*`)`. For a ternary relation, such as *x owes y dollars to z*, the representation problem is more difficult.

We are thus led to a generalization of the notion of *attribute*. As an alternative, a symbol may be declared to be a *predicate* and used to represent a relation of an unspecified number of arguments. Attributes and predicates are both called *relation symbols*. A *proposition* is a list whose members are a relation symbol followed by arguments. A proposition associated with an attribute must have exactly two arguments, but a predicate proposition may have any number of arguments. A proposition that resides in the database is called an *assertion*. Thus, a slot value is just an assertion pertaining to an attribute.

For example, if the symbol `OWES` is recognized as a predicate, then it may be used to represent the ternary relation mentioned above, and the statement *DAVID owes 15 dollars to DONALD* may be asserted by

```
@assert (owes david 15 donald)
```

to which the system responds by creating the new datum

```
Assertion OWES-1    (IN)
  (OWES DAVID 15 DONALD)
```

There is no restriction on the appearance of variables in the argument list of an assertion that is attached to a predicate. (In a slot value, a variable may appear only as the first argument, and then its type must be a subtype of the defining class of the attribute.) An assertion is classified as *general* or *particular* according to whether or not it contains any variables. The variables in a general assertion are understood to be universally quantified. Thus, the assertion

```
General Assertion OWES-2    (IN)
  (OWES ?U:UNDERGRADUATE 15 ?F:FACULTY)
```

represents the statement *each undergraduate owes 15 dollars to each faculty member*.

4 Backward Inference

An *instance* of a proposition is a second proposition that results from the first by performing some set of variable substitutions. When a proposition is presented to the Proteus theorem prover as a *goal*, it attempts to derive an instance of it from the knowledge in the database.

One way in which it might succeed is to unify the goal with an assertion. The process of unification amounts to finding the most general common instance of two propositions. If a goal is unifiable with an assertion that is `IN`, then the resulting instance of the goal is returned as the result of the proof. For example, if the database of Fig. 7 contains

```
Class Slot Value DESCRIPTION-5    (IN)
  (DESCRIPTION ?X:STUDENT IDEALISTIC)
```

then the goal `(DESCRIPTION ?X:EMPLOYEE ?Y)` could succeed by returning the instance `(DESCRIPTION ?X:(GRADER TA) IDEALISTIC)`.

A goal may also be proved with the use of a *backward rule*. A backward rule is composed of a proposition, called its *consequent*, and one or more *antecedents*. A rule represents the belief that any instance of its consequent is true whenever any compatible instances of its antecedents are true. If a goal is unified with the consequent, then the corresponding instances of the antecedents become subgoals — recursively proving all of these subgoals completes the proof of the original goal. This process is known as *backward chaining* or *goal-directed inference*, and is the basis of Prolog [Clocksin and Mellish 1981] and other logic programming systems.

For example, in order to derive a value for *MICHAEL*'s `UNCLE` slot, the goal `(UNCLE ?X MICHAEL)` may be unified with the consequent of the rule

```
Backward Rule UNCLE-1    (IN)
```

```
(UNCLE ?X ?Y)
 <--
(PARENT ?X ?Z)
(BROTHER ?Z ?Y)
```

creating the subgoals `(PARENT MICHAEL ?Z)` and `(BROTHER ?Z ?Y)`. Suppose that the first of these is matched with the consequent of

```
Backward Rule PARENT-1    (IN)
  (PARENT ?X ?Y)
   <--
  (MOTHER ?X ?Y)
```

and is thus replaced by the subgoal `(MOTHER MICHAEL ?Y)`, which is matched with

```
Instance Slot Value MOTHER-7 (IN) (MOTHER MICHAEL SUZY)
```

The second subgoal, which becomes `(BROTHER SUZY ?Y)`, may then be derived from

```
 Instance Slot Value BROTHER-23    (IN)
  (BROTHER SUZY DAVID)
```

The instance `(UNCLE MICHAEL DAVID)` of the original goal is thereby proved by backward chaining.

A Proteus predicate may alternatively be defined in Lisp, rather than by rules and assertions. This provides the user the full power of Lisp for knowledge representation and also allows access to Common Lisp system functions, as in

```
Backward Rule POWER-OF-TWO-2    (IN)
  (POWER-OF-TWO ?X:FIXNUM)
   <--
  (<= 1 ?X)
  (EVENP ?X)
  (POWER-OF-TWO (/ ?X 2))
```

Here the predicate `POWER-OF-TWO` is defined in terms of the two predicates `<=` and `EVENP`, both of which are defined by Common Lisp. The first subgoal produced by this rule succeeds if the function `<=` returns true for the arguments `1` and the binding of `?X`. Note that the interface between Lisp and the rule system also allows function calls to be embedded in antecedents, as in the third antecedent above.

An antecedent may also take the form of a proposition preceded by the symbol `UNLESS`, as in

```
Backward Rule HAS-CHILD-1    (IN)
  (HAS-CHILD ?X)
   <--
  (MOTHER ?Y ?X)
```

```
UNLESS (ADULT ?Y)
```

When an antecedent of this type is processed, the system attempts to prove the proposition that follows the `UNLESS` (under the current variable bindings). If this proof attempt fails, then the subgoal succeeds; if the proof succeeds, then the subgoal fails.

When a proposition proved by backward chaining is explicitly added to the database as an assertion, it receives a justification that is constructed upon examination of the proof. Thus, the proposition derived in the first example of this section would result in

```
Instance Slot Value UNCLE-2
  (UNCLE MICHAEL DAVID)
```

which would acquire a justification with `IN`-list `(UNCLE-1 PARENT-1 MOTHER-7 BROTHER-23)`, i.e., all the data involved in the proof, and `OUT`-list `()`.

Nonmonotonic dependencies are constructed from proofs that involve `UNLESS` antecedents. For example, if the proposition `(HAS-CHILD SUZY)` were derived from `Backward Rule HAS-CHILD-1` and `Assertion MOTHER-1` above, and

```
Assertion HAS-CHILD-5
  (HAS-CHILD SUZY)
```

were created as a result, then the `IN`-list of its justification would be `(HAS-CHILD-1 MOTHER-7)`, but the justification would also reflect the dependency of the derivation on the failure to prove `(ADULT MICHAEL)`. This is done by making the `OUT`-list `(ADULT-2)`, where

```
Assertion ADULT-2    (OUT)
  (ADULT MICHAEL)
```

is an unjustified assertion, created for the purpose of this justification (unless it already existed). If `(ADULT MICHAEL)` were to be asserted later, then `Assertion ADULT-2` would become `IN`, and `HAS-CHILD-5` would go `OUT` as it should.

The case of an `UNLESS` goal with unbound variables presents a new problem. Suppose, for example, that we have a predicate `ORPHAN` with an associated rule

```
Backward rule ORPHAN-1     (IN)
  (ORPHAN ?X)
   <--
  unless (PARENT ?X ?Z)
```

Then the goal `(ORPHAN ANNIE)` will succeed if `(PARENT ANNIE ?Z)` fails. In this case, the new justification for

```
Assertion ORPHAN-2:
  (ORPHAN ANNIE)
```

should contain only `ORPHAN-1` in its `IN`-list, but there is no assertion, general or particular, which could be placed in the `OUT`-list to record the nonmonotonic dependency. This problem is solved by the introduction of a new datatype:

```
Failed Goal PARENT-2    (OUT)
  (PARENT ANNIE ?Z)
```

A *failed goal* is a datum that is created only in this situation. When a proof succeeds as a result of a failure to prove a proposition that follows `UNLESS` in an antecedent, the proposition that failed is inserted in the database without justification as a failed goal. It represents the belief that some instance of the proposition is true. That is, any variables in a failed goal are understood to be existentially quantified. The failed goal PARENT-2 above, which represents the belief that *ANNIE* has some parent, would appear in the `OUT`-list of the justification of `ORPHAN-2`.

If some instance of a failed goal is asserted at any time, the system automatically creates a monotonic dependency of the failed goal on the assertion. Thus, if the assertion `ORPHAN-2` were justified as described above, and

```
Assertion PARENT-3:
  (PARENT ANNIE WARBUCKS)
```

were later to become `IN`, then the failed goal `PARENT-2` would also be forced `IN` and hence `ORPHAN-2` would go `OUT`.

5 Forward Inference

A backward rule has effect only when it is relevant to a goal being processed by the system. The insertion of a backward rule, therefore, affects only the implicit informational content of the database, without causing new assertions to be added explicitly. Consider, for example, the backward rule `PARENT-1` of Section 5, which states that all mothers are parents. In the presence of

```
Instance Slot Value MOTHER-7    (IN)
  (MOTHER MICHAEL SUZY)
```

this rule enlarges the implicit database to include the proposition `(PARENT SUZY MICHAEL)` without actually creating a new assertion.

The same logical implication expressed by `PARENT-1` could alternatively be represented as a *forward rule*:

```
Forward Rule MOTHER-11
  (MOTHER ?X ?Y)
   -->
  (PARENT ?X ?Y)
```

While the two rules are logically equivalent, they are used quite differently. The forward rule takes effect not when a goal matches its consequent `(PARENT ?X ?Y)`, but rather when an assertion matches its antecedent `(MOTHER ?X ?Y)`. In this event (assuming the new assertion is `IN`), another assertion, representing the corresponding instance of `(PARENT ?X ?Y)`, is automatically added to the database. The `PARENT` assertion is then justified by the `MOTHER` assertion and the rule. Thus, when the match between `MOTHER-7` and `MOTHER-11` is discovered, the result is a new datum

```
Instance Slot Value PARENT-4   (IN)
  (PARENT MICHAEL SUZY)
```

which is justified with an `IN`-list `(MOTHER-7 MOTHER-11)` and aa`OUT`-list `NIL`. This process is known as *forward chaining* or *data-directed inference.*

A forward rule may have any number of antecedents and consequents. Antecedents have the same form as those of backward rules. When a new assertion is unified with an antecedent of a forward rule, the set of remaining antecedents is presented to the backward inference system as goals. For each simultaneous proof of these goals, a *firing* of the rule occurs, i.e., its consequents are processed.

A consequent of a forward rule may be either a proposition or a Lisp form. When a rule is fired, an instance of each propositional consequent (corresponding to the derived instances of the antecedents) is asserted. The justification for this assertion is constructed from the data involved in the derivation of the antecedents, as described in Section 5. Each Lisp consequent is simply evaluated, with variables evaluating to their bindings.

Suppose, for example, that the database contains

```
Forward rule PATIENT-1     (IN)
```

```
(PATIENT ?X)
(SHOULD-TAKE ?X ?Y)
 -->
(UNDER-TREATMENT ?X)
(FORMAT T "Prescription for ~A:  ~A" ?X ?Y)
```

when

```
Assertion PATIENT-2     (IN)
  (PATIENT BILL)
```

is added. The first antecedent of `PATIENT-2` is matched with `PATIENT-1`, triggering an attempt to prove `(SHOULD-TAKE BILL ?Y)`. Suppose that the instance `(SHOULD-TAKE BILL ASPIRIN)` is derived. Then after

```
Assertion SHOULD-TAKE-1     (IN)
  (SHOULD-TAKE BILL ASPIRIN)
```

is added to the database, the rule `PATIENT-1` fires:

```
Assertion UNDER-TREATMENT-1     (IN)
  (UNDER-TREATMENT BILL)
```

is added, justified by `PATIENT-1`, `PATIENT-2`, and `SHOULD-TAKE-1`, and

```
Prescription for BILL:  ASPIRIN
```

is printed.

Some thought is required in determining whether a given implication should be represented as a forward rule or a backward rule. A backward rule offers the advantage of increasing the inherent knowledge of a system without incurring the expense (in both time and space) of creating new assertions. It may be necessary, however, for this knowledge to be represented explicitly in order for it to take some desired effect.

Suppose, for example, that the assertion

```
Assertion ORPHAN-3    (IN)
  (ORPHAN GEORGE)
```

is added as a result of the rule `ORPHAN-1` of Section 5. Then the `OUT`-list of its justification contains a datum corresponding to the last antecedent of the rule,

```
Failed Goal PARENT-5   (OUT)
  (PARENT GEORGE ?Z)
```

which was unprovable at the time `ORPHAN-3` was created. Suppose that

```
Assertion MOTHER-12    (IN)
  (MOTHER GEORGE MARY)
```

were asserted later. It would then be desirable for `PARENT-5` to come `IN` and for `ORPHAN-3` to go `OUT`. The backward rule `PARENT-1`, however, could not cause this to occur. Although an instance of `PARENT-5` would become provable, that instance would not be discovered. It would probably be preferable in this case to code the rule in the form of the forward rule `MOTHER-11` instead. This would produce a new assertion

```
Assertion PARENT-6   (IN)
  (PARENT GEORGE MARY)
```

on which `PARENT-5` would become monotonically dependent, and `ORPHAN-3` would go `OUT` as desired.

References

[Aït-kaci, et al. 1985] Aït-kaci, H., R. Boyer, and R. Nasr, *An Encoding Technique for the Efficient Implementation of Type Inheritance*, MCC Technical Report AI-109-85.

[Aït-kaci and Nasr 1985] Aït-kaci, H., and R. Nasr, *LOGIN: A Logic Programming Language with Built-in Inheritance*, MCC Technical Report AI-109-85.

[Charniak, et al. 1980] Charniak, E., C. K. Riesbeck, and D. V. McDermott, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, N.J., 1980.

[Clocksin and Mellish 1981] Clocksin, W. F. & C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[Doyle 1979] Doyle, J., A Truth Maintenance System, in *Artificial Intelligence*, Vol.12 No.3, 1979.

[Goodwin 1986] Goodwin, J. W., *WATSON: A Dependency Directed Inference System*, Research Report LiTH-IDA-R-84-10, Computer and Information Science Dept., Linkøping University.

[Petrie 1987] Petrie, C., *Revised Dependency-Directed Backtracking for Default reasoning*, Proceedings of AAAI-87, Seattle, 1987.

[Poltrock 1986] Poltrock, S., D. Steiner, & N. Tarlton Graphic Interfaces for Knowledge-Based System Development, in *Proceedings of ACM/SIGCHI*, Boston, MA, 1986.

[Russinoff 1985a] Russinoff, D., *An Algorithm for Truth Maintenance*, MCC Technical Report AI-062-85.

[Russinoff 1985b] Russinoff, D., *A Nonmonotonic Inference System*, MCC Technical Report AI-062-85.