28

Meta-Level Knowledge

Randall Davis and Bruce G. Buchanan

This chapter explores a number of issues involving representation and use of what we term *meta-level knowledge*, or knowledge about knowledge.[1] It begins by defining the term, then exploring a few of its varieties and considering the range of capabilities it makes possible. Four specific examples of meta-level knowledge are described, and a demonstration given of their application to a number of problems, including interactive transfer of expertise and the "intelligent" use of knowledge. Finally, we consider the long-term implications of the concept and its likely impact on the design of large programs. The context of this work is the TEIRESIAS program discussed in Chapter 9. In the earlier chapter we focused on the *use of* TEIRESIAS for knowledge acquisition. Here we focus on the classification and types of knowledge *used by* TEIRESIAS.

In the most general terms, meta-level knowledge is knowledge about knowledge. Its primary use here is to enable a program to "know what it knows," and to make multiple uses of its knowledge. As mentioned in Chapter 9, the program is not only able to use its knowledge directly, but may also be able to examine it, abstract it, reason about it, or direct its application.

This chapter discusses examples of meta-level knowledge classified along two dimensions: (i) specificity character (*representation-specific* vs. *domain-specific*), and (ii) source (*user-supplied* vs. *derived*). Representation-specific meta-level knowledge involves supplying a program with a store of knowledge dealing with the *form* of its representations, in particular, their design and organization. Traditionally, this design and organization infor-

This chapter is an expanded and edited version of a paper originally appearing in *Proceedings of the Fifth IJCAI*, 1977, pp. 920–928. Used by permission of International Joint Conferences on Artificial Intelligence, Inc.; copies of the *Proceedings* are available from William Kaufmann, Inc., 95 First Street, Los Altos, CA 94022.

[1]Following standard usage, knowledge about objects and relations in a particular domain will be referred to as *object-level knowledge*.

I. Knowledge about contents of rules in the knowledge base—Rule Models
II. Knowledge about syntax
  Of the representation of objects—Schemata
  Of predicate functions—Function Templates
III. Knowledge about strategies—Meta-Rules

FIGURE 28-1 Classification of meta-level knowledge in TEIRESIAS.

mation is present in a system only implicitly, for example, in the way a particular segment of code accesses data or the way a chunk of knowledge is encoded. Type declarations are a small step toward more explicit specification of this information, especially as they are used in extended data types and record structures. As we discuss below, this sort of information, along with a range of other facts about representation design, can be employed quite usefully if it is made explicit and made available to the system.

Domain-specific meta-level knowledge contains information dealing with the *content* of object-level knowledge, independent of its particular encoding. It might involve any kind of useful information about a chunk of knowledge, including its likely utility, range of applicability, speed or space requirements, capabilities, and side effects. The two examples given here deal with forms of meta-level knowledge that (i) offer information about global patterns and trends in the content of object-level knowledge, and (ii) provide strategic information, i.e., knowledge about how best to use other knowledge.

The examples described below also illustrate the difference between user-supplied and derived meta-level knowledge. The former is of course obtained from the user; the latter is derived by the system on the basis of information it already has. The user-supplied variety is used as a source for knowledge that the system could not have deduced on its own; the derived form allows the system to uncover useful characteristics of the knowledge base and to make maximal use of knowledge it already has.

As will become clear below, meta-level knowledge makes possible a number of interesting capabilities. The representation-specific variety supports knowledge acquisition, provides assistance on knowledge base maintenance, and makes possible multiple distinct uses of a single chunk of knowledge. The domain-specific type provides a site for embedding information about the most effective use of knowledge and can have a significant impact on both the efficiency displayed by a system and its level of performance. The examples also demonstrate that the source of the meta-level knowledge has an impact on system performance. In particular, the derived variety is shown to make possible a very simple but potentially useful form of closed-loop behavior.

We examine below the four instances of meta-level knowledge used by TEIRESIAS (shown in Figure 28-1) and review for each (i) the basic idea, explaining why it is a form of meta-level knowledge; (ii) a specific instance, detailing the information it contains; (iii) an example of how that information is used to support knowledge base construction, maintenance, or use; and (iv) the other capabilities it makes possible, including a limited form of self-knowledge.

28.1 Rule Models

28.1.1 Rule Models as Empirical Abstractions of the Knowledge Base

As described in Chapter 9, a rule model is an abstract description of a subset of rules, built from empirical generalizations about those rules. It is used to characterize a "typical" member of the subset and is composed of four parts. First, a list of examples indicates the subset of rules from which this model was constructed.

Next, a description characterizes a typical member of the subset. Since we are dealing in this case with rules composed of premise-action pairs, the description currently implemented contains individual characterizations of a typical premise and a typical action. Then, since the current representation scheme used in those rules is based on associative triples, we have chosen to implement those characterizations by indicating (a) which attributes "typically" appear in the premise (and in the action) of a rule in this subset and (b) correlations of attributes appearing in the premise (and in the action).[2] Note that the central idea is the concept of *characterizing a typical member of the subset.* Naturally, that characterization looks different for subsets of rules than it does for procedures, theorems, frames, etc. But the main idea of characterization is widely applicable and not restricted to any particular representational formalism.

The two remaining parts of the rule model are pointers to models describing more general and more specific rule models covering larger or smaller subsets of rules. The set of models is organized into a number of tree structures, each of the general form shown in Figure 28-2. This structure determines the subsets for which models will be constructed. At the root of each tree is the model made from all the rules that conclude about <attribute>; below this are two models dealing with all affirmative and all negative rules; and below this are models dealing with rules that affirm or deny specific values of the attribute. There are several points to note here. First, these models are not hardwired into the system, but are instead formed by TEIRESIAS on the basis of the current contents of the knowledge base. Second, whereas the knowledge base contains object-level rules about a specific domain, the rule models contain information about those

[2]Both of these are constructed via simple statistical thresholding operations.

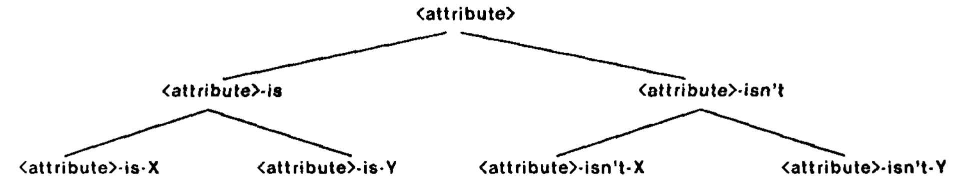

FIGURE 28-2 Organization of the rule models.

rules, in the form of empirical generalizations. As such, they offer a global overview of the regularities in the rules. The rule models are thus an example of derived, domain-specific meta-level knowledge.

28.1.2 Rule Model Example

Figure 28-3 shows an example of a rule model, one that describes the subset of rules concluding affirmatively about the area for an investment.[3] (Since not all details of implementation are relevant here, this discussion will omit some.) As indicated above, there is a list of rules from which this model was constructed, descriptions characterizing the premises and actions, and pointers to more specific and more general models. Each characterization in the description is shown split into its two parts, one concerning the presence of individual attributes and the other describing correlations. The first item in the premise description, for instance, indicates that "most" rules about the area of investment mention the attribute RETURNRATE in their premises; when they do mention it, they "typically" use the predicate functions SAME and NOTSAME; and the "strength," or reliability, of this piece of advice is 3.83.

The fourth item in the premise description indicates that when the attribute RETURNRATE (rate of return) appears in the premise of a rule in this subset, the attribute TIMESCALE "typically" appears as well. As before, the predicate functions are those usually associated with the attributes, and the number is an indication of reliability.

28.1.3 Use of Rule Models in Knowledge Acquisition

Use of the rule models to support knowledge acquisition occurs in several steps. First, as noted in Chapter 9, our model of knowledge acquisition is one of interactive transfer of expertise in the context of a shortcoming in

[3]These examples were generated by substituting investment terms for medical terms in examples from TEIRESIAS using MYCIN's medical knowledge.

```
MODEL FOR RULES CONCLUDING AFFIRMATIVELY ABOUT INVESTMENT AREA

EXAMPLES        ((RULE116 .33)
                 (RULE050 .70)
                 (RULE037 .80)
                 (RULE095 .90)
                 (RULE152 1.0)
                 (RULE140 1.0))

DESCRIPTION
  PREMISE       ((RETURNRATE SAME NOTSAME 3.83)
                 (TIMESCALE SAME NOTSAME 3.83)
                 (TREND SAME 2.83)

                 ((RETURNRATE SAME) (TIMESCALE SAME) 3.83)
                 ((TIMESCALE SAME) (RETURNRATE SAME) 3.83)
                 ((BRACKET SAME) (FOLLOWS NOTSAME SAME) (EXPERIENCE SAME) 1.50))

  ACTION        ((INVESTMENT-AREA CONCLUDE 4.73)
                 (RISK CONCLUDE 4.05)

                 ((INVESTMENT-AREA CONCLUDE) (RISK CONCLUDE) 4.73))

MORE-GENL       (INVESTMENT-AREA)

MORE-SPEC       (INVESTMENT-AREA-IS-UTILITIES)
```

FIGURE 28-3 Example of a rule model.

the knowledge base. The process starts with the expert challenging the system with a specific problem and observing its performance. If the expert believes its results are incorrect, there are available a number of tools that will allow him or her to track down the source of the error by selecting the appropriate rule model. For instance, if the problem is a missing rule in the knowledge base to conclude about the appropriate area for an investment, then TEIRESIAS will select the model shown in Figure 28-3 as the appropriate one to describe the rule it is about to acquire. Note that the selection of a specific model is in effect an expression by TEIRESIAS of its *expectations* concerning the new rule, and the generalizations in the model become predictions about the likely content of the rule.

At this point the expert types in the new rule (Figure 28-4), using the vocabulary specific to the domain. (In all traces, computer output is in mixed upper and lower case, while user responses are in boldface capitals.)

As mentioned in Chapter 9 and further described in Chapter 18, English text is understood by allowing keywords to suggest partial interpretations and intersecting those results with the expectations provided by the selection of a particular rule model. We thus have a data-directed process (interpreting the text) combined with a goal-directed process (the predictions made by the rule model). Each contributes to the end result, but it is their combination that is effective. TEIRESIAS displays the results of

```
The new rule will be called RULE383
        If:     1 - THE CLIENT'S INCOME TAX BRACKET IS 50%
            and 2 - THE CLIENT IS FOLLOWING UP ON MARKET TRENDS CAREFULLY
            and 3 -
        Then:   1 - THERE IS EVIDENCE (.8) THAT THE INVESTMENT AREA SHOULD BE HIGH
                    TECHNOLOGY
            and 2 -

This may take a few moments . . .
```

FIGURE 28-4 New rule as stated by the expert.

this initial interpretation of the rule (Figure 28-5). If there are mistakes (as there are in this case), a rule editor is available to allow the expert to indicate necessary changes. This is easily accomplished, since TEIRESIAS can often make an effective second choice by determining the likely source of error in its initial guess.

Once the expert is satisfied that TEIRESIAS has correctly understood what was said, it is the system's turn to see if *it* is satisfied with the content of the rule. The main idea is to use the rule model to see how well this new rule "fits into" the system's model of its knowledge—i.e., does it "look like" a typical rule of the sort expected?

If the expert agrees to the inclusion of a new clause, TEIRESIAS attempts to create it (Figure 28-6). The system relies on the context of the current dialogue (which indicates that the clause should deal with the amount of the client's investment experience) and the fact that the rule must work for this case or it won't fix the bug (it is not shown here, but earlier in the interaction the expert indicated that the client had a *moderate* amount of experience). TEIRESIAS' guess is not necessarily correct, of course, since the desired clause may be more general, but it is at least a plausible attempt.

It should be noted that there is nothing in this concept of "second-guessing" that is specific to the rule models as they are currently designed, or indeed to associative triples of rules as a knowledge representation. The most general and fundamental point was mentioned above—testing to see how something "fits into" the system's model of its knowledge. At this point

```
This is my understanding of your rule:

RULE383
     IF:     1) The client's income-tax bracket is 50%, and
             2) The market has followed a upward trend recently, and
             3) The client manages his assets carefully
     THEN:   There is evidence (.8) that the area of the investment should be high-technology
```

FIGURE 28-5 TEIRESIAS' first interpretation of the rule shown in Figure 28-4.

```
I hate to criticize, Randy, but did you know that most rules about what the area of investment might be, that mention-
        the income-tax bracket of the client, and
        how closely the client follows the market
ALSO mention -
        [A] - the amount of investment experience of the client
Shall I try to write a clause to account for [A]?
++** Y

How about -
        [A] The amount of investment experience of the client is moderate
Ok?
++** Y
```

FIGURE 28-6 TEIRESIAS' suggestion of an additional clause to the new rule based on the rule model shown in Figure 28-3.

the system might perform any kind of check for violations of any established prejudices about what the new chunk of knowledge should look like. Additional kinds of checks for rules might concern the strength of the inference, the number of clauses in the premise, etc. In general, this "second-guessing" process can involve any characteristic that the system may have "noticed" about the particular knowledge representation in use.

Automatic generation of rule models has several interesting implications, since it makes possible a synthesis of the ideas of model-based understanding and learning by experience. While both of these have been developed independently in previous AI research, their combination produces a novel sort of feedback loop: rule acquisition relies on the set of rule models to effect the model-based understanding process; this results in the addition of a new rule to the knowledge base; and this in turn triggers recomputation of the relevant rule model(s).

Note, first, that performance on the acquisition of a subsequent rule may be better, because the system's "picture" of its knowledge base has improved—the rule models are now computed from a larger set of instances, and their generalizations are more likely to be valid. Second, since the relevant rule models are recomputed each time a change is made to the knowledge base, the picture they supply is kept constantly up to date, and they will at all times be an accurate reflection of the shifting patterns in the knowledge base.

Finally, and perhaps most interesting, the models are not hand-tooled by the system architect or specified by the expert. They are instead formed by the system itself, and formed as a result of its experience in acquiring rules from the expert. Thus, despite its reliance on a set of models as a basis for understanding, TEIRESIAS' abilities are not restricted by a pre-*existing set of models.* As its store of knowledge grows, old models can become more accurate, new models will be formed, and the system's stock of knowledge about its knowledge will continue to expand.

28.2 Schemata

28.2.1 The Need for Knowledge About Representations

As data structures go beyond the simple types available in most programming languages to extended data types defined by the user, they typically become rather complex. Large programs may have numerous structures that are complex in both their internal organization and their interrelationships with other data types in the system. Yet information about these details may be scattered in comments in system code, in documents and manuals maintained separately, and in the mind of the system architect. This presents problems to anyone changing the system. Consider, for example, the difficulties encountered in such a seemingly simple problem as adding a new instance of an existing data type to a large program. Just finding all of the necessary information can be a major task, especially for someone unfamiliar with the system.

One particularly relevant set of examples comes from the numerous approaches to knowledge representation that have been tried over the years. While the emphasis in discussions of predicate calculus, semantic nets, production rules, frames, etc., has naturally concerned their respective conceptual power, at the level of implementation each of these carries problems of data structure management.

Our second example of meta-level knowledge, then, is of the representation-specific variety and involves describing to a system a range of information about the representations it employs. The main idea here is, first, to view every knowledge representation in the system as an extended data type and to write explicit descriptions of them. These descriptions should include all of the information about structure and interrelations that is often widely scattered. Next, we devise a language in which all of this can be put in machine-comprehensible terms and write the descriptions in those terms, making this store of information available to the system. Finally, we design an interpreter for the language, so that the system can use its new knowledge to keep track of the details of data structure construction and maintenance.

The approach is based on the concept of a *data structure schema*, a device that provides a framework in which representations can be specified. The framework, like most, carries its own perspectives on its domain. One point it emphasizes strongly is the detailed specification of many kinds of information about representations. It attempts to make this specification task easier by providing ways of organizing the information and a relatively high-level vocabulary for expressing it.

Schema hierarchy: indicates categories of representations and their organization

Individual schema: describes structure of a single representation

Slot names: (the schema building blocks) describe implementation conventions

FIGURE 28-7 Levels of knowledge about representations.

28.2.2 Schema Example

There are three levels of organization of the information about representations (Figure 28-7). At the highest level, a schema hierarchy links the schemata together, indicating what categories of data structure exist in the system and the relationships among them. At the next level of organization are individual schemata, the basic units around which the information about representations is organized. Each schema indicates the structure and interrelationships of a single type of data structure. At the lowest level are the slot names (and associated structures) from which the schemata are built; these offer knowledge about specific conventions at the programming language level. Each of these three levels supplies a different sort of information; together they comprise an extensive body of knowledge about the structure, organization, and implementation of the representations.

The hierarchy is a generalization hierarchy (Figure 28-8) that indicates the global organization of the representations. It makes extensive use of the concept of inheritance of properties, so that a particular schema need represent only the information not yet specified by schemata above it in the hierarchy. This distribution of information also aids in making the network extensible.

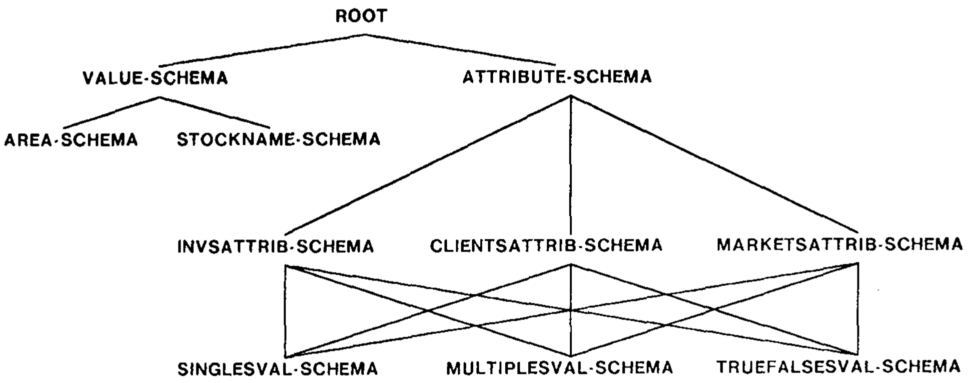

FIGURE 28-8 Part of the schema hierarchy.

Each schema contains several different types of information:

1. the structure of its instances,
2. interrelationships with other data structures,
3. a pointer to all current instances,
4. inter-schema organizational information, and
5. bookkeeping information.

Figure 28-9 shows the schema for a stock name; information corresponding to each of the categories listed above is grouped together. The first five lines in Figure 28-9 contain structure information and indicate some of the entries on the property list (PLIST) of the data structure that represents a stock name. The information is a triple of the form

<slot name> <blank> <advice>

The slot name labels the "kind" of thing that fills the blank and serves as a point around which much of the "lower-level" information in the system is organized. The blank specifies the format of the information required, while the advice suggests how to find it. Some of the information needed may be domain-specific, and hence must be requested from the expert. But some of it may concern completely internal conventions of representation, and hence should be supplied by the system itself, to insulate the domain expert from such details. The advice provides a way of indicating which of these situations holds in a given case.

```
STOCKNAME-SCHEMA
  PLIST        [( INSTOF STOCKNAME-SCHEMA                                    GIVENIT
                  SYNONYM (KLEENE (1 0) < ATOM >)                            ASKIT
                  TRADEDON (KLEENE (1 1 2) <(MARKET-INST FIRSTYEAR-INST)>)   ASKIT
                  RISKCLASS CLASS-INST                                       ASKIT
                 CREATEIT]

  RELATIONS    ( (AND* STOCKNAMELIST HILOTABLE)
                 (OR* CUMVOTINGRIGHTS)
                 (XOR* COMMON PFD CUMPFD PARTICPFD)
                 ((OR* PFD CUMPFD PARTICPFD) PFDRATETABLE)
                 ((AND* CUMPFD) OMITTEDDIVS) )

  INSTANCES    (AMERICAN-MOTORS AT&T . . . XEROX ZOECON)

  FATHER       (VALUE-SCHEMA)

  OFF-SPRING   NIL

  DESCR        "the STOCKNAME-SCHEMA describes the format for a stock name"
  AUTHOR       DAVIS
  DATE         1115
  INSTOF       (SCHEMA-SCHEMA)
```

FIGURE 28-9 Schema for a stock name.

The next five lines in the schema (under RELATIONS) indicate its interrelations with other data structures in the system. The main point here is to provide the system architect with a way of making explicit all of the data structure interrelationships on which the design depends. Expressing them in a machine-accessible form makes it possible for TEIRESIAS to take over the task of maintaining them, as explained below.

The schemata also keep a list of all current instantiations of themselves (under INSTANCES), primarily for use in maintaining the knowledge base. If the design of a data structure requires modification, it is convenient to have a pointer to all current instances to ensure that they are similarly modified.

The next two lines (FATHER and OFF-SPRING) contain organizational information indicating how the stock name schema is connected to the schema hierarchy.

Finally, there are four slots for bookkeeping information to help keep track of a large number of data structures: each structure is tagged with the date of creation and author, along with a free-text description supplied by the author. In addition, each structure has a pointer to the schema of which it is an instance (note in this case that it is the schema itself that is the data structure being described by this information).

28.2.3 Use of Schemata in Knowledge Acquisition

Use of the schemata in knowledge acquisition relies on several ideas:

- Information in the schema is viewed as a guide to creating a new instance of the representation it describes.
- That guidance is supplied by (a) the structure description information, which is in the form of a prototype to be instantiated, and (b) the relations information, which is interpreted as pointers to a number of structures that may require updating to ensure that necessary data structure interrelations are maintained.
- It is this instantiation and interpretation process that drives the knowledge transfer dialogue.
- The advice present in the schema adds a level of sophistication to the dialogue.

For instance, suppose in the process of adding a new rule to the system the expert mentions a stock the system hasn't heard about yet. Learning about the new stock (i.e., creating a new instance of the concept of stock name) becomes a subproblem in the task of acquiring the new rule. The schema for the concept of stock name is retrieved and used as the basis for a dialogue that requests the necessary information from the expert. An abbreviated version of the dialogue is shown in Figure 28-10.

```
The new rule will be called RULE384
  If  1 - THE DESIRED RATE OF RETURN ON THE INVESTMENT IS GREATER THAN 10%
      2 - THE AMOUNT OF INVESTMENT EXPERIENCE OF THE CLIENT IS MODERATE
      3 - THE AREA OF THE INVESTMENT IS NATURAL RESOURCE DEVELOPMENT
  Then
    1 - THERE IS EVIDENCE (.5) THAT THE NAME OF THE STOCK TO INVEST IN
        IS GEORGIA PACIFIC

Sorry, but I don't understand
       THERE IS EVIDENCE (.5) THAT THE NAME OF THE STOCK TO INVEST IN
       IS GEORGIA PACIFIC
because these words are unknown: GEORGIA PACIFIC
```

{TEIRESIAS finds it cannot generate any interpretations of the text, so it reprints the original line and specifies where the trouble is. This makes it possible for the expert to figure out what it is that is new to the system.}

```
Would you care to try rephrasing that?
++** Y
  (Please use *'s to mark what you think might be a new attribute or a
   new value of an existing attribute.)

      1 - THERE IS EVIDENCE (.5) THAT THE NAME OF THE STOCK TO INVEST
          IN IS * GEORGIA PACIFIC *
```

{While the language in this example is very simple, there are more complex cases where it would be quite hard to decide from the context what sort of new item the expert had in mind. To avoid the necessity for more sophisticated natural language handling, the expert is asked to mark off the new word or phrase with asterisks.}

```
It looks as though
       GEORGIA PACIFIC
refers to a new value of an attribute, correct?
++** Y
. . . is it the name of a stock?
++** Y
```

{. . . and with the new item clearly marked, the system can then do some very simple pattern matching to decide what's going on.}

```
Now please tell me a few things about it . . .
```

{TEIRESIAS retrieves the schema for a stock, and "reads" it to find out what it needs to know about any stock.}

```
On what markets is GEORGIA-PACIFIC traded?
The answer should be in the form of 1 or more of the following-
       <market name> <year first traded>
[Type 1 set to a line, then an empty line when done.]
++** NYSE 1945
++** PCE 1968
++**
```

{Each of the questions from the system is prompted by the attempt to fill in some component of the data structure being built, according to the representation conventions indicated by the schema.}

```
What is the current risk classification of GEORGIA-PACIFIC?
The answer should be in the form of a
       <risk classification>
++** AMEX
```

FIGURE 28-10 Excerpt of dialogue in which the system learns a new rule.

```
  Sorry, but the following are invalid -
      AMEX is not a recognized <risk classification>
  Please answer again [use the same answer if you really meant it].
+ + ** BLUE-CHIP
+ + **
      {TEIRESIAS now begins working on updating other data structures in response to the addition of the new
      stock name.}
Should this new item be added to-
      1 - the list of common stocks, or
      2 - the list of preferred stocks, or
      3 - the list of cumulative preferred stocks, or
      4 - the list of participating preferred stocks
Select one of the above
+ + ** 3
[Adding GEORGIA-PACIFIC to CUMULATIVEPFD]
      {One of these tasks involves adding the new stock to one of four mutually exclusive lists. (Several other
      necessary updates are also performed; these have been edited out here for brevity.)}
Ok, done with GEORGIA-PACIFIC now. . . Back to the rule.
      {Having finished acquiring information about the new stock, the system returns to parsing the new rule, and
      with some help from the expert gets the correct result [the parsing process is omitted here, see Davis
      (1977) for a detailed example].}
This may take a few moments.
This is my understanding of your rule:
  If  1 - the desired rate of return for the investment is greater than 10%
      2 - the amount of investment experience of the client is moderate
      3 - the area of investment is natural-resource-development
  Then
      1 - there is evidence (.5) that the name of the stock to choose is georgia-pacific
```

FIGURE 28-10 continued

28.2.4 Other Uses of Schemata

The preceding subsection showed one instance of using schemata for maintenance of the knowledge base. They help ensure that one change to the knowledge base (adding a new instance of a known representation) will not violate necessary relationships between data structures. The schemata also support other capabilities. Besides being useful in maintaining the knowledge base, they offer a convenient mechanism for organizing and implementing data structure access and storage functions.

One of the ideas behind the design of the schemata is to use them as points around which to organize knowledge. The information about structure and interrelationships described above, for instance, is stored this way. In addition, access and storage information is also organized in this fashion. By generalizing the advice concept slightly, it is possible to effect all data structure access and storage requests in the appropriate schema. That is, code that needs to access a particular structure "sends" an access request, and the structure "answers" by providing the requested item.[4] This offers the well-known advantage of insulating the implementation of a data structure from its logical design. Code that refers only to the latter is far easier to maintain in the face of modifications to data structure implementation.

28.3 Function Templates

Associated with each predicate function in the system is a *template,* a list structure that resembles a simplified procedure declaration (Figure 28-11). It is representation-specific, indicating the order and generic type of the arguments in a typical call of that function. Templates make possible two interesting parallel capabilities: code generation and code dissection. Templates are used as a basis for the simple form of code generation alluded to in Chapter 9. Although details are beyond the scope of this chapter [see Davis (1976)], code generation is essentially a process of "filling in the blanks": processing a line of text in a new rule involves checking for keywords that implicate a particular predicate function, and then filling in its template on the basis of connotations suggested by other words in the text.

Function	*Template*
SAME	(object attribute value)

FIGURE 28-11 Template for the predicate function SAME.

Code dissection is accomplished by using the templates as a guide to extracting any desired part of a function call. For instance, as noted earlier, TEIRESIAS forms the rule models on the basis of the current contents of the knowledge base. To do this, it must be able to pick apart each rule to determine the attributes to which it refers. This could have been made possible by requiring that every predicate function use the same function call format (i.e., the same number, type, and order of arguments), but this would be too inflexible. Instead, we allow every function to describe its own calling format via its template. To dissect a function call, then, we need only retrieve the template for the relevant function and then use the template as a guide to dissecting the remainder of the form. The template in Figure 28-11, for instance, indicates that the *attribute* would be the second item after the function name. This same technique is also used by TEIRESIAS' explanation facility, where it permits the system to be quite precise in the explanations it provides.

[4]This was suggested by the perspective taken in work on SMALLTALK (Goldberg and Kay, 1976) and ACTORS (Hewitt et al., 1973). This style of writing programs has come to be known as object-oriented programming.

This approach also offers a useful degree of flexibility. The introduction of a new predicate function, for instance, can be totally transparent to the rest of the system, as long as its template can be written in terms of the available set of primitives such as attribute, value, etc. The power of this approach is limited primarily by this factor and will succeed to the extent that code can be described by a relatively small set of such primitive descriptors. While more complex syntax is easily accommodated (e.g., the template can indicate nested function calls), more complex semantics are more difficult (e.g., the appearance of multiple attributes in a function template can cause problems).

Finally, note that the templates also offer a small contribution to system maintenance. If it becomes necessary to modify the calling sequence of a function, for instance, we can edit just the template and have the system take care of effecting analogous changes to all current invocations of the function.

28.4 Meta-Rules

28.4.1 Meta-Rules—Strategies to Guide the Use of Knowledge

A second form of domain-specific meta-level knowledge is *strategy knowledge* that indicates how to use other knowledge. This discussion considers strategies from the perspective of *deciding which knowledge to invoke next* in a situation where more than one chunk of knowledge may be applicable. For example, given a problem solvable by either heuristic search or problem decomposition, a strategy might indicate which technique to use, based on characteristics of the problem domain and nature of the desired solution. If the problem decomposition technique were chosen, other strategies might be employed to select the appropriate decomposition from among several plausible alternatives.

This view of strategies is useful because many of the paradigms developed in AI admit (or even encourage) the possibility of having several alternative chunks of knowledge be plausibly useful in a single situation (e.g., production rules, logic-based languages, etc.). When a set of alternatives is large enough (or varied enough) that exhaustive invocation is infeasible, some decision must be made about which should be chosen. Since the performance of a program will be strongly influenced by the intelligence with which that decision was made, strategies offer an important site for the embedding of knowledge in a system.

A MYCIN-like system invokes rules in a simple backward-chaining fashion that produces an exhaustive depth-first search of an AND/OR goal tree. If the program is attempting, for example, to determine which stock would make a good investment, it retrieves all the rules that make a conclusion about that topic (i.e., they mention STOCKNAME in their action clauses). It then invokes each one in turn, evaluating each premise to see if the conditions specified have been met. The search is exhaustive because the rules are inexact: even if one succeeds, it was deemed to be a wisely conservative strategy to continue to collect all evidence about a subgoal.

The ability to use an exhaustive search is of course a luxury, and in time the base of rules may grow large enough to make this infeasible. At this point some choice would have to be made about which of the plausibly useful rules should be invoked. *Meta-rules* were created to address this problem. They are rules *about* object-level rules and provide a strategy for pruning or reordering object-level rules before they are invoked.

28.4.2 Examples of Meta-Rules

Figure 28-12 shows four meta-rules for MYCIN (reverting to medicine again for the moment). The first of them says, in effect, that in trying to determine the likely identities of organisms from a sterile site, rules that base their identification on other organisms from the same site are not likely to be successful. The second indicates that when dealing with pelvic abscess, organisms of the class *Enterobacteriacae* should be considered before gram-positive rods. The third and fourth are like the second in that they reorder relevant rules before invoking them.

It is important to note the character of the information conveyed by meta-rules. First, note that in all cases we have a rule that is making a conclusion about other rules. That is, where object-level rules conclude about the medical (or other) domain, meta-rules conclude about object-level rules. These conclusions can (in the current implementation) be of two forms. As in the first meta-rule, they can make deductions about the likely utility of certain object-level rules, or as in the second, they can indicate a partial ordering between two subsets of object-level rules.

Note also that (as in the first example) meta-rules make conclusions about the *utility* of object-level rules, not about their *validity*. That is, METARULE001 does not indicate circumstances under which some of the object-level rules are invalid [or even "very likely (.9)" to be invalid]. It merely says that they are likely not to be *useful;* i.e., they will probably fail, perhaps only after requiring extensive computation to evaluate their preconditions. This is important because it has an impact on the question of distribution of knowledge. If meta-rules did comment on validity, it might make more sense to distribute the knowledge in them, i.e., to delete the meta-rule and just add another premise clause to each of the relevant object-level rules. But since their conclusions concern utility, it does not make sense to distribute the knowledge.

Adding meta-rules to the system requires only a minor addition to MYCIN's control structure. As before, the system retrieves the entire list

```
METARULE001
IF  1) the culture was not obtained from a sterile source, and
    2) there are rules which mention in their premise a previous
       organism which may be the same as the current organism
THEN it is definite (1.0) that each of them is not going to be useful.

PREMISE:  ($AND (NOTSAME CNTXT STERILESOURCE)
                (THEREARE OBJRULES (MENTIONS CNTXT PREMISE
                'SAMEBUG) SET1))
ACTION:  (CONCLIST SET1 UTILITY NO TALLY 1.0)


METARULE002
IF  1) the infection is a pelvic-abscess, and
    2) there are rules which mention in their premise
       enterobacteriaceae, and
    3) there are rules which mention in their premise gram-positive rods,
There is suggestive evidence (.4) that the former should be done before
    the latter.

PREMISE:  ($AND (SAME CNTXT PELVIC-ABSCESS)
                (THEREARE OBJRULES(MENTIONS CNTXT PREMISE
                            ENTEROBACTERIACEAE) SET1)
                (THEREARE OBJRULES(MENTIONS CNTXT PREMISE GRAMPOS-RODS)
                            SET2))
ACTION:  CONCLIST SET1 DOBEFORE SET2 TALLY .4)


METARULE003
IF  1) there are rules which do not mention the current goal in
       their premise
    2) there are rules which mention the current goal in their
       premise
THEN it is definite that the former should be done before the latter.

PREMISE:  ($AND(THEREARE OBJRULES ($AND (DOESNTMENTION FREEVAR
                       ACTION CURGOAL))SET1)
                (THEREARE OBJRULES ($AND (MENTIONS FREEVAR PREMISE
                       CURGOAL)SET2))
ACTION:  (CONCLIST SET1 DOBEFORE SET2 1000)


METARULE004
IF  1) there are rules which are relevant to positive cultures, and
    2) there are rules which are relevant to negative cultures
THEN it is definite that the former should be done before the latter.

PREMISE:  ($AND(THEREARE OBJRULES ($AND (APPLIESTO FREEVAR POSCUL))
                            SET1)
                (THEREARE OBJRULES ($AND (APPLIESTO FREEVAR NEGCUL))
                            SET2))
ACTION:  (CONCLIST SET1 DOBEFORE SET2 1000)
```

FIGURE 28-12 Four meta-rules for MYCIN.

of rules relevant to the current goal (call the list L). But before attempting to invoke them, it first determines if there are any meta-rules relevant to the goal.[5] If so, these are invoked first. As a result of their actions, we may obtain a number of conclusions about the likely utility and relative ordering of the rules in L. These conclusions are used to reorder or shorten L, and the revised list of rules is then used. Viewed in tree-search terms, the current implementation of meta-rules can either prune the search space or reorder the branches of the tree.

28.4.3 Guiding the Use of the Knowledge Base

There are several points to note about encoding knowledge in meta-rules. First, the framework it presents for knowledge organization and use appears to offer a great deal of leverage, since much can be gained by adding to a system a store of (meta-level) knowledge about which chunk of object-level knowledge to invoke next. Considered once again in tree terms, we are talking about the difference between a "blind" search of the tree and one guided by heuristics. The advantage of even a few good heuristics in cutting down the combinatorial explosion of tree search is well known. Thus, where earlier sections were concerned about adding more object-level knowledge to improve performance, here we are concerned with giving the system more information about how to use what it already knows. Consider, too, that the definition of intelligence includes appropriate use of information. Even if a store of (object-level) information is not large, it is important to be able to use it properly. Meta-rules provide a mechanism for encoding strategies that can make this possible.

Second, the description given in the preceding subsection has been simplified in several respects for the sake of clarity. It discusses the augmented control structure, for example, in terms of two levels. In fact, there can be an arbitrary number of levels, each serving to direct the use of knowledge at the next lower level. That is, the system retrieves the list (L) of object-level rules relevant to the current goal. Before invoking this, it checks for a list (L′) of first-order meta-rules that can be used to reorder or prune L, etc. Recursion stops when there is no rule set of the next higher order, and the process unwinds, each level of strategies advising on the use of the next lower level. We can gain leverage at this higher level by encoding heuristics that guide the use of heuristics. That is, rather than adding more heuristics to improve performance, we might add more information at the next higher level about effective use of existing heuristics.

[5]That is, are there meta-rules directly associated with that goal? Meta-rules can also be associated with other objects in the system, but that is beyond the scope of this chapter. The issues of organizing and indexing meta-rules are covered in more detail elsewhere (Davis, 1976; 1978).

The judgmental character of the rules offers several interesting capabilities. It makes it possible, for instance, to write rules that make different conclusions about the best strategy to use and then rely on the underlying model of confirmation (Shortliffe and Buchanan, 1975) to weigh the evidence. That is, the strategies can "argue" about the best rule to use next, and the strategy that "presents the best case" (as judged by the confirmation model) will win out.

Next, recall that the basic control structure of the performance program is a depth-first search of the AND/OR goal tree sprouted by the unwinding of rules. The presence of meta-rules of the sort shown in Figure 28-12 means that this tree has an interesting characteristic at each node: when the system has to choose a path, there may be information stored that advises about the best path to take. There may therefore be available an extensive body of knowledge to guide the search, but that knowledge is not embedded in the code of a clever search algorithm. It is instead organized around the specific objects that form the nodes in the tree; i.e., instead of a smart algorithm, we have a "smart tree."

Finally, there are several advantages associated with the use of strategies that are goal-specific, explicit, and imbedded in a representation that is the same as that of the object-level knowledge. The fact that strategies are *goal-specific,* for instance, makes it possible to specify precise heuristics for a given goal, without imposing any overhead on the search for any other goals. That is, there may be a number of complex heuristics describing the best kinds of rules to use for a particular goal, but these will cause no computational overhead except in the search for that goal.

The fact that they are *explicit* means a conceptually cleaner organization of knowledge and an ease of modification of established strategies. Consider, for instance, alternative means of achieving the sort of partial ordering specified by the second meta-rule. There are several alternative schemes by which this could be accomplished, involving appropriate modifications to the relevant object-level rules and slight changes to the control structure. Such schemes, however, share several faults that can be illustrated by considering one such approach: an agenda with multiple priority levels like the one proposed in Bobrow and Winograd (1977).

In an agenda-driven system, rules are put on an agenda rather than dealt with in the form of a linear list of relevant rules in a partial ordering. Partial ordering could be accomplished simply by setting the priority of some rules higher than that of others; rules in subset A, for instance, might get priority 6, while those in subset B are given priority 5. But this technique presents two problems: it is both opaque and likely to cause bugs. It will not be apparent from looking at the code, for instance, *why* the rules in A were given a higher priority than that of the rules in B. Were they more likely to be useful, or is it desirable that those in A precede those in B no matter how useful they may be? Consider also what happens if, before we get a chance to invoke any of the rules in A, an event occurs that makes it clear that their priority ought to be reduced (for reasons unrelated to the desired partial ordering). If the priority of only the rules in A is adjusted, a bug arises, since the desired relative ordering may be lost.

The problem is that this approach tries to reduce a number of different, incommensurate factors to a single number, *with no record of how that number was reached.* Meta-rules offer one mechanism for making these sorts of considerations explicit, and for leaving a record of why a set of processes has been queued in a particular order. They also make subsequent modifications easier, since all of the information is in one place—changing a strategy can be accomplished by editing the relevant meta-rule, rather than by searching through a program for all the places where priorities have been set to effect that strategy.

Lastly, the use of a *uniform encoding of knowledge* makes the treatment of all levels the same. For example, second-order meta-rules require no machinery in excess of that needed for first-order meta-rules. It also means that all the explanation and knowledge acquisition capabilities developed for object-level rules can be extended to meta-rules as well. The first of these (explanation) has been done and works for all levels of meta-rules. Adding this to TEIRESIAS' explanation facility makes possible an interesting capability: in addition to being able to explain what it did, the system can also explain *how it decided to do what it did.* Knowledge in the strategies has become accessible to the rest of the system and can be explained in just the same fashion. We noted above that adding meta-level knowledge to the system was quite distinct from adding more object-level knowledge, since strategies contain information of a qualitatively different sort. Explanations based on this information are thus correspondingly different as well.

28.4.4 Broader Implications of Meta-Rules

The concept of strategies as a mechanism for deciding which chunk of knowledge to invoke next can be applied to a number of different control structures. We have seen how it works in goal-directed scheme, and it functions in much the same way with a data-directed process. In the latter case meta-rules offer a way of controlling the depth and breadth of the implications drawn from any new fact or conclusion. Pursuing this further, we can imagine making the decision to use a data- or goal-directed process itself as an issue to be decided by a collection of appropriate meta-rules. At each point in its processing, the system might invoke one set of meta-rules to choose a control structure, then use another set to guide that control structure. This can be applied to many control structures, demonstrating the range of applicability of the basic concept of strategies as a device for choosing what to do next.

28.4.5 Content-Directed Invocation

If meta-rules are to be used to select from among plausibly useful object-level rules, they must have some way of referring to the object-level rules. The mechanism used to effect this reference has implications for the flexibility and extensibility of the resulting system. To see this, note that the meta-rules in Figure 28-12 refer to the object-level rules by *describing* them and effect this description by direct examination of content. For instance, METARULE001 refers to *rules that mention in their premises previous organisms that may be the same as the current organism,* which is a description rather than an equivalent list of rule names. The set of object-level rules that meet this description is determined at execution time by examining the source code of the rules. That is, the meta-rule "goes in and looks" for the relevant characteristic, using the function templates as a guide to dissecting the rules. We have termed this *content-directed invocation.*

Part of the utility of this approach is illustrated by its advantages over using explicit lists of object-level rules. If such lists were used, then tasks would require extensive amounts of bookkeeping. After an object-level rule had been edited, for instance, we would have to check all the strategies that name it, to be sure that each such reference was still applicable to the revised rule. With content-directed invocation, however, these tasks require no additional effort, since the meta-rules effect their own examination of the object-level rules and will make their own determination of relevance.

28.5 Conclusions

We have reviewed four examples of meta-level knowledge and demonstrated their application to the task of building and using large stores of domain-specific knowledge. This has showed that supplying the system with a store of information about its representations makes possible a number of useful capabilities. For example, by describing the structure of its representations (schemata, templates), we make possible a form of transfer of expertise, as well as a number of facilities for knowledge base maintenance. By supplying strategic information (meta-rules), we make possible a finer degree of control over use of knowledge in the system. And by giving the system the ability to derive empirical generalizations about its knowledge (rule models), we make possible a number of useful abilities that aid in knowledge transfer.

The examples reviewed above illustrate a number of general ideas about knowledge representation and use that may prove useful in building large programs. We have, first, the notion that knowledge in programs should be made explicit and accessible. Use of production rules to encode the object-level knowledge is one example of this, since knowledge in them may be more accessible than that embedded in the code of a procedure. The schemata, templates, and meta-rules illustrate the point also, since each of them encodes a form of information that is, typically, either omitted entirely or at best is left implicit. By making knowledge explicit and accessible, we make possible a number of useful abilities. The schemata and templates, for example, support the forms of system maintenance and knowledge acquisition described above. Meta-rules offer a means for explicit representation of the decision criteria used by the system to select its course of action. Subsequent "playback" of those criteria can then provide a form of explanation of the motivation for system behavior [see Davis (1976) for examples]. That behavior is also more easily modified, since the information on which it is based is both clear (since it is explicit) and retrievable (since it is accessible). Finally, more of the system's knowledge and behavior becomes open to examination, especially by the system itself.

Second, there is the idea that programs should have access to their own representations. To put this another way, consider that over the years numerous representation schemes have been proposed and have generated a number of discussions of their respective strengths and weaknesses. Yet, in all these discussions, one entity intimately concerned with the outcome has been left uninformed: the program itself. What this suggests is that we ought to describe to the program a range of information about the representations it employs, including such things as their structure, organization, and use.

As noted, this is easily suggested but more difficult to do. It requires a means of describing both representations and control structures, and the utility of those descriptions will be strongly dependent on the power of the language in which they are expressed. The schemata and templates are the two main examples of the partial solutions we have developed for describing representations, and both rely heavily on the idea of a task-specific high-level language—a language whose conceptual primitives are task-specific. The main reason for using this approach is to make possible what we might call "top-down code understanding." Traditionally, efforts at code understanding [e.g., Waldinger and Levitt (1974), Manna (1969)] have attempted to assign meaning to the code of some standard programming language. Rather than take on this sizable task, we have used task-specific languages to make the problem far easier. Instead of attempting to assign semantics to ordinary code, we assigned a "meaning" to each of the primitives in the high-level language and represented it in one or more informal ways. Thus, for example, ATTRIBUTE is one of the primitives in the "language" in which templates are written; its meaning is embodied in procedures associated with it that are used during code generation and dissection [see Davis (1976) for details].

This convenient shortcut also implies a number of limitations. Most importantly, the approach depends on the existence of a finite number of "mostly independent" primitives. This means a set of primitives with only a few, well-specified interactions between them. The number of interactions should be far less than the total possible, and interactions that do occur should be uncomplicated (as, for example, the interaction between the concepts of attribute and value).

But suppose we could describe to a system its representations? What benefits would follow? The primary thing this can provide is a way of effecting multiple uses of the same knowledge. Consider, for instance, the multitude of ways in which the object-level rules have been used. They are executed as code in order to drive the consultation (see Part Two); they are viewed as data structures, and dissected and abstracted to form the rule models (Parts Three and Nine); they are dissected and examined in order to produce explanations (Part Six); they are constructed during knowledge acquisition (Part Three); and, finally, they are reasoned about by the meta-rules (Part Nine).

It is important to note here that the feasibility of such multiplicity of uses is based less on the notion of production rules *per se* than on the availability of a representation with a *small grain size* and a *simple syntax and semantics.* "Small" modular chunks of code written in a simple, heavily stylized form (though not necessarily a situation-action form) would have done as well, as would have any representation with simple enough internal structure and of manageable size. The introduction of greater complexity in the representation, or the use of a representation that encoded significantly larger "chunks" of knowledge, would require more sophisticated techniques for dissecting and manipulating representations than we have developed thus far. But the key limitations are size and complexity of structure, rather than a specific style of knowledge encoding.

Two other benefits may arise from the ability to describe representations. We noted earlier that much of the information necessary to maintain a system is often recorded in informal ways, if at all. If it were in fact convenient to record this information by describing it to the program itself, then we would have an effective and useful repository of information. We might see information that was previously folklore or informal documentation becoming more formalized and migrating into the system itself. We have illustrated above a few of the advantages this offers in terms of maintaining a large system.

This may in turn produce a new perspective on programs. Early scarcity of hardware resources led to an emphasis on minimizing machine resources consumed, for example, by reducing all numeric expressions to their simplest form by hand. More recently, this has meant a certain style of programming in which a programmer spends a great deal of time thinking about a problem first, trying to solve as much as possible by hand, and then abstracting out only the very end product of all of that effort to be embodied in the program. That is, the program becomes simply a way of manipulating symbols to provide "the answer," with little indication left of what the original problem was or, more importantly, what knowledge was required to solve it.

But what if we reversed this trend, and instead viewed a program as a place to store many forms of knowledge about both the problem and the proposed solution (i.e., the program itself)? This would apply equally well to code and data structures and could help make possible a wider range of useful capabilities of the sort illustrated above.

One final observation. As we noted at the outset, interest in knowledge-based systems was motivated by the belief that no single domain-independent paradigm could produce the desired level of performance. It was suggested instead that a large store of domain-specific (object-level) knowledge was required. We might similarly suggest that this too will eventually reach its limits and that simply adding more object-level knowledge will no longer, by itself, guarantee increased performance. Instead, it may be necessary to focus on building stores of meta-level knowledge, especially in the form of strategies for effective use of knowledge. Such "meta-level knowledge–based" systems may represent a profitable future direction for research.