META-LEVEL KNOWLEDGE: OVERVIEW AND APPLICATIONS

Randall Davis and Bruce G. Buchanan
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

We define the concept of meta-level knowledge, and illustrate it by briefly reviewing four examples that have been described in detail elsewhere [2-5]. The examples include applications of the idea to tasks such as transfer of expertise from a domain expert to a program, and the maintenance and use of large knowledge bases. We explore common themes that arise from these examples, and examine broader implications of the idea, in particular its impact on the design and construction of large programs.

This work was supported in part by the Bureau of Health Sciences Research and Evaluation of HEW under Grant HS-01544 and by the Advanced Research Projects Agency under ARPA Order 2494. It was carried out on the SUMEX-AIM Computer System, supported by the NIH under Grant RR-00785. The views expressed are solely those of the author.

{1} Introduction

The representation and use of knowledge has been a central problem in AI research. A range of different encoding techniques have been developed, along with a number of approaches to applying knowledge. Most of the effort to date, however, has concentrated on representing and manipulating knowledge about a specific domain of application, like game-playing ([14]), natural language understanding ([15], [19]), speech understanding ([8], [11]), chemistry ([7]), etc.

This paper explores a number of issues involving representation and use of what we term *meta-level knowledge*, or knowledge about knowledge. 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 guiding the use of knowledge. Finally, we consider the long term implications of the concept and its likely impact on the design of large programs.

{2} Meta-level knowledge

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. That is, 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. To see in general terms how this can be accomplished, imagine taking some of the available representation techniques and turning them in on themselves, using them to describe their own encoding and use of knowledge. The result is a system with a store of both knowledge about the domain (the object level knowledge), and knowledge about its representations (the meta-level knowledge).

{3} Background

Some early efforts in AI involved the search for a single problem solving paradigm that would be both powerful and widely (or even universally) applicable. By the late 1960's it became clear that a single such paradigm was at best elusive, and that high (i.e., near human level) performance on non-trivial tasks required large stores of domain specific knowledge. A number of such knowledge-based systems have been developed and the methodology applied to a wide range of tasks, including speech understanding [11], algebraic symbol manipulation [12], and chemistry [7]. Because of the magnitude of the task of assembling the knowledge base for these systems, the accumulation, management and use of large stores of task specific knowledge has itself become a significant research problem.

It was this problem that provided the context for the development and exploration of meta-level knowledge reported here. The examples described below are all aimed toward the three aspects of the problem noted just above (knowledge accumulation, management, and use):

Schemata (Section 4.1) and rule models (Section 4.2) support accumulation of knowledge via interactive transfer of expertise from a human expert to the knowledge base of the system.

The schemata, along with the function templates (Section 4.3), provide a mechanism for handling some aspects of knowledge base maintenance.

Finally, meta-rules (Section 4.4) are applied to the problem of guiding the use of knowledge by offering a means of expressing strategies.

All of these are part of the TEIRESIAS system [2-5], an INTERLISP program designed to function as an assistant in the construction of high performance programs. A key element in this construction process is the transfer of expertise from a human expert to the program. Since the domain expert often knows nothing about programming, his interaction with the performance program usually requires a human programmer as intermediary. We have sought to create in TEIRESIAS a program to supply the same sort of assistance as that provided by the programmer, in order to remove the programmer from the loop.

We view the interaction between the domain expert and the performance program in terms of a teacher who continually challenges a student with new problems to solve, and carefully observes the student's performance. The teacher may interrupt to request a justification of some particular step the student has taken in solving the problem, or may challenge the final result. This may uncover a fault in the student's knowledge of the subject, and result in the transfer of information to correct it.

Figure 1 below shows the overall architecture of the sort of program TEIRESIAS is designed to help construct. The *knowledge base* is the program's store of task specific knowledge that makes possible high performance. The *inference engine* is an interpreter that uses the knowledge base to solve the problem at hand.

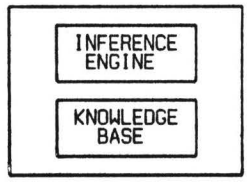

Figure 1 - architecture of the performance program

The main point of interest in this very simple design is the explicit division between these two parts of the program. This division allows us to assign the human expert the task of augmenting the knowledge base of a program whose control structure (inference engine) is assumed both appropriate and debugged. The question of *how* knowledge is to be encoded and used is settled by the selection of one or more of the available representations and control structures. The expert's task is to enlarge *what* it is the program knows. If all of the control structure information has been kept in the inference engine, then we can engage the domain expert in a discussion of the knowledge base and be assured that the discussion will have to deal only with issues of domain specific expertise (rather than with questions of programming and control structures).

In this discussion we will assume the knowledge base contains information about selecting an investment in the stock market; the performance program thus functions as an investment consultant.[1] Knowledge is in the form of a collection of associative triples (attribute, object, value) which characterize the domain, and approximately 400 inference rules built from them (Figure 2). Each rule is a single "chunk" of domain specific information indicating an action (in this case a conclusion) which is justified if the conditions specified in the premise are fulfilled.

```
RULE027
If   [1] the time-scale of the investment is long-term,
     [2] the desired return on the investment is
         greater than 10%,
     [3] the area of the investment is not known,
then there is evidence (.4) that the name of the stock
     to invest in is AT&T.

PREMISE    ($AND (SAME OBJCT TIMESCALE LONG-TERM)
                 (GREATER OBJCT RETURNRATE 10)
                 (NOTKNOWN OBJCT INVESTMENT-AREA))
ACTION     (CONCLUDE OBJCT STOCK-NAME AT&T .4)
```

Figure 2 - inference rule (English and LISP forms)

{4} Types of meta-level knowledge

We examine below four examples of meta-level knowledge, and review for each (*i*) the general idea; (*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. Figure 3 summarizes the type of information contained in each of the four examples.

KNOWLEDGE ABOUT	IS ENCODED IN
**********	**********
representation of objects	schemata
representation of functions	function templates
inference rules	rule models
reasoning strategies	meta-rules

Figure 3 - four types of meta-level knowledge

{4.1} Example 1: Schemata

{4.1.1} Introduction: 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 which are complex in both their internal organization and their interrelationships with other data types in the system. That is, the design and organization of data structures in any sizable system often involves a non-trivial store of detailed information. Yet such information is typically widely scattered, perhaps throughout comments in system code, in documents and manuals maintained separately, and in the mind of the system architect.

This presents a problem to someone who wants to make any sort of change to the system. Consider, for example, the difficulties typically 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 representations which 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 has presented a non-trivial problem in data structure management.

The second example of meta-level knowledge 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 write explicit descriptions of each of them. These descriptions should include all the information about structure and interrelations that was noted earlier as 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.

This is of course easily said and somewhat harder to do. It involves answering a number of difficult questions concerning the content of the required knowledge, and concerning how that information should be represented and used. This paper gives an overview of the answers, details can be found in [2] and [3]. The discussion here demonstrates briefly that the relevant knowledge includes information about the structure and interrelations of representations, and shows that it can be used as the basis for a form of knowledge acquisition.

The approach is based on the concept of a *data structure schema*, a device which provides a framework in which representations can be specified. This 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.

{4.1.2} Schema example

There are three levels of organization of the information about representations (Figure 4). At the highest level, a schema hierarchy links the schemata together, indicating what categories of data structures exist in the system and the relationships between them. At the next level of organization are the individual schemata, the basic unit 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 slotnames (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 compose an extensive body of knowledge about the structure, organization, and implementation of the representations.

schema hierarchy - indicates categories of representations and their organization
individual schema - describes structure of a single representation
slotnames - the schema building blocks, describe implementation conventions

Figure 4

The hierarchy is a generalization hierarchy 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 (see [2] for examples and further details).

Each individual 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
5) bookkeeping information

Figure 5 shows the schema for a stock name; information corresponding to each of the categories listed above is grouped together.

```
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)
            (XOR* COMMON PFD CUMPFD PARTICPFD)
            ((OR* PFD CUMPFD) PFDRATETABLE)
            ((AND* CUMPFD) OMITTEDDIVS) )

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

 FATHER        (VALUE-SCHEMA)
 OFFSPRING     NIL

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

Figure 5 - schema for a stock name

The first five lines in Figure 5 contain structure information, and indicate some of the entries on the property list (PLIST) of the data structure which represents a stock name. The information is a triple of the form

<slotname> <blank> <advice>

The *slotname* labels the "kind" of things which 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 may concern solely 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 each case.

The next five lines in the schema 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 upon which his 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, 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 insure that they are similarly modified.

The next two lines contain organizational information indicating how the the stockname schema is connected to the schema hierarchy.

Finally, there is four lines of bookkeeping information that helps in keeping 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 which is the data structure being described by this information).

{4.1.3} Schemata: use in knowledge acquisition

Use of the schemata for 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
  - the structure description information, which is in the form of a prototype to be instantiated, and
  - the *relations* information, which is interpreted as pointers to a number of structures which may require updating to insure that necessary data structure interrelations are maintained.
- it is this instantiation and interpretation process that drives the knowledge transfer dialog.
- the *advice* present in the schema adds a level of sophistication in the dialog.

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 *stock name*) becomes a subproblem in the task of acquiring the new rule. The stock name schema is retrieved, and used as the basis for a dialog that requests the necessary information from the expert. Detailed examples are found in [2] and [3]; an abbreviated version of the dialog is shown below. (In all traces, computer output is in mixed upper and lower case in this font, while user responses are in BOLDFACE CAPITALS, and commentary is {inside braces}.).

```
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 the trouble.}

```
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, he 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 a name of a stock?
++** Y
```

{The system can then do some very simple pattern matching.}

```
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. A translated form of the "blank" is printed to guide the expert; it is then used to parse his answer and insure syntactic validity.}

```
What is the current risk classification of GEORGIA-
PACIFIC? The answer should be in the form of a
          <risk classification>
++** AMEX
  Sorry, but the following are invalid -
  AMEX is not a recognized <risk classification>
  Please answer again.
++** BLUE-CHIP
```

{TEIRESIAS now beings 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 prefered stocks, or
     4 - the list of participating prefered 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, see [3] for details).}

```
Ok, done with GEORGIA-PACIFIC now... Back to the rule.
```

{Having finished acquiring 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 [4] 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 the investment is natural-
         resource-development
 Then
     1 - there is evidence (.5) that the name of the
         stock to choose is georgia-pacific
```

{4.1.4} Schemata: other uses

The schemata also support a number of other capabilities. They are useful in maintaining the knowledge base, for instance, and offer a convenient mechanism for organizing and implementing data structure access and storage functions.

The data structure updating demonstrated in the previous section is one instance of their maintenance capabilities. This updating helps to insure that one change to the knowledge base (adding a new instance of representation) will not violate necessary relationships between data structures.

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 via the appropriate schema. That is, code which wants to access a particular structure "sends" an access request, and the structure "answers" by providing the requested item[2]. This offers the well known advantages of insulating the implementation of a data structure from its logical design. Code which refers only to the latter is far easier to maintain in the face of modifications to data structure implementation.

While they have not yet been implemented, two other interesting uses of the schemata appear possible. First, straightforward extensions to the current system should support a more complex form of knowledge base maintenance. Suppose, for instance, it became necessary to modify the representation of a stock, i.e., we want to edit the stock name schema. It should be possible to have TEIRESIAS "watch" as the schema is modified and then carry out the same sequence of modifications on each of the current instances of the schema. Where new information was required (e.g., if new structure descriptors were added to the schema) the system could prompt for the appropriate entry for each instance. While major redesigns would be more difficult to carry out in this fashion, a number of common modifications could be accommodated, easing the task of making changes to structures in the knowledge base.

Second, the schema also appear to make possible a limited form of introspection. If the information in the *relations* slot were made accessible via simple retrieval routines, this would make it possible to answer questions like *What else in the system will be affected if I add a new instance of this data structure?* or *What are all the other structures that are related to this one?* This would be a useful form of on-line documentation.

{4.2} Example 2: Rule models

{4.2.1} Rule models as empirical abstractions of the knowledge base

In reviewing the rules in the knowledge base, a number of regularities become apparent. In particular, rules about a single topic tend to have characteristics in common -- there are "ways" of reasoning about a given topic. This idea of patterns of reasoning has been given a formal (statistical) definition, and provides the basis for the automated construction of a set of empirical generalities about the knowledge base: the rule models.

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 in this sense is similar to the structures used in [20]), 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 (action).[3]

Note that the central idea is the concept of *characterizing a typical member of the subset*. Naturally, that characterization would look different for subsets of rules, procedures, theorems, etc. But the main idea of characterization is widely applicable and not restricted to any particular representational formalism.

The two other parts of the rule model are pointers to models describing more general and more specific subsets of rules. The set of models is organized into a number of tree structures. These structures determine the subsets for which models will be constructed. At the root of each tree is the model made from all the rules which conclude about <attribute>, below this are two models dealing with all affirmative and all negative rules, and below this are models dealing with rules which 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 content of the knowledge base. Second, where the rules in the knowledge base contain object level information about a specific domain, the rule models contain information about those rules, in the form of empirical generalizations. As such they offer a global overview of the regularities in the rules, and may possibly reflect useful trends in the reasoning of the expert from whom those rules were acquired.

{4.2.2} Rule model example

Figure 6 shows an example of a rule model, one that describes the subset of rules concluding affirmatively about the area for an investment. (Since not all of the details of implementation are relevant here, this discussion will omit some. See [2] for a full explanation.) As indicated above, there is a list of the rules from which this model was constructed, descriptions characterizing the premise and the action, 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 what the area of an investment should be mention the attribute *rate of return* in their premise; 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 (see [2] for precise definitions of the quoted terms).

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

```
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 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 6 - example of a rule model

{4.2.3} Rule models: use in knowledge acquisition

Use of the rule models to support knowledge acquisition occurs in several steps. First, as noted above, our model of knowledge acquisition is one of interactive transfer of expertise in the context of a shortcoming in the knowledge base. The process starts with the expert challenging the system with a specific problem and observing its performance. If he believes its results are incorrect, there are available a number of tools that will allow him to track down the source of the error (see [2] for details). TEIRESIAS keeps track of this debugging process, and responds to the discovery of the source of the error by selecting the appropriate rule model. For instance, if the problem is a rule missing from the knowledge base that concludes about the appropriate area for an investment, then TEIRESIAS will select the model shown in Figure 6 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 7), using the vocabulary specific to the domain, and expressing it as much as possible in the associative triple format. TEIRESIAS's problem now is to try to understand what the expert has said. As is traditional, "understanding" is determined by converting the text into an internal representation (like that shown in Figure 2), then converting this back into English and requesting approval from the expert.

Since understanding natural language is known to be difficult, we have taken a simpler approach. The basic idea is to allow the text to "suggest" interpretations via a simple keyword-based approach, and to intersect 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 the combination of them that is effective. Details of this process are described in [2] and [4].

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 -
```

Figure 7

TEIRESIAS displays the results of this initial interpretation of the rule (Figure 8). If there are mistakes (as there are in this case), a rule editor is available to allow the expert to indicate required 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.

```
This is my understanding of your rule:
RULE383
If  1) The client's income-tax bracket is 50%,
    2) The market has followed a upward trend recently
    3) The client manages his assets carefully
Then   there is evidence (.8) that the area of the
       investment should be high-technology
```

Figure 8

Once the expert is satisfied that TEIRESIAS has correctly understood what he 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 in" to the system's model of its knowledge -- i.e., does it "look like" a typical rule of the sort expected?

In the current implementation, the presence of a partial match between the new rule and the generalizations in the rule model triggers a response from TEIRESIAS. Recall the last line of the premise description in the rule model of Figure 6:

((BRACKET SAME) (FOLLOWS SAME) (EXPERIENCE SAME) 1.50))

This indicates that when the tax BRACKET of the client appears in the premise of a rule of this sort, then how closely he FOLLOWS the market, and how much investment EXPERIENCE he has typically appear as well. Note that the new rule has the first two of these, but is missing the last, and TEIRESIAS points this out.

```
I hate to criticize, Randy, but did you know that most
rules about what the area of a 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 9

If the expert agrees to the inclusion of a new clause, TEIRESIAS attempts to create it. The system relies on the context of the current dialog (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 (earlier in the interaction [not shown] the expert indicated that the client had a *moderate* amount of experience). TEIRESIAS's 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" which is specific to the rule models as they are currently designed, or indeed to associative triples or rules as a knowledge representation. The most general and fundamental point was mentioned above -- testing to see how something "fits in" to the system's model of its knowledge. At this point 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, number of clauses in the premise, etc. Different checks might be devised for other knowledge encoding schemes.

The automatic generation of the rule models by TEIRESIAS 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 prompts the recomputation of the relevant rule model(s).

Note first that performance on the acquisition of the next 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's abilities are not restricted by the 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. This appears to be a novel capability for a model-based system.

{4.2.4} Rule models: other capabilities

As a form of meta-level knowledge, the rule models give the system a picture of its own knowledge. The system can, for instance, "read" a rule model to the user, supplying an overview of the information in part of the knowledge base. This may suggest global trends in the knowledge of the expert who assembled the knowledge base, and thus helps to make clear the overall approach of the system to a given topic (for examples see [2]).

{4.3} Example 3: Function templates

Associated with each predicate function in the system is a *template*, a list structure which resembles a simplified procedure declaration (Figure 10). It indicates the order and generic type of the arguments in a typical call of that function, and makes possible very simple versions of two interesting, parallel capabilities: code generation and code dissection.

```
FUNCTION        TEMPLATE
SAME            (OBJ ATTRIBUTE VALUE)
```

Figure 10 - template for the predicate function SAME

The template is used as the basis for the simple form of code generation alluded to in Section {4.2.3}. While details are beyond the scope of this paper (see [2]), 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.

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 (i.e., the template for the CAR of the form), and then use that as a guide to dissecting the remainder of the form. The template in Figure 10, for instance, indicates that the *attribute* would be the CADDR of the form. This same technique is also used by TEIRESIAS's explanation facility, where it permits the system to be quite precise in the explanations it provides (see [2] for details).

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 like *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 accomodated (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).

{4.4} Example 4: Meta-rules

{4.4.1} Strategies to guide the use of knowledge

Meta-rules embody strategies -- 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 can be useful because many of the paradigms developed in AI admit (or even encourage) the possibility of having several alternative chunks of knowledge plausibly useful in a single situation (e.g., production rules, PLANNER-like languages, etc.). Faced with a set of alternatives large enough (or varied enough) that exhaustive invocation becomes 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 is made, strategies offer an important site for the embedding of knowledge in a system.

This type of guidance can be especially useful in the sort of rule-based performance program that TEIRESIAS is designed to help build. The rules in this system are invoked 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 which make a conclusion about that topic (i.e., they mention **STOCK-NAME** in their action). 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 wiseiy 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. As 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.

{4.4.2} Meta-rules: examples

Figure 11 below shows two meta-rules. The first of them says, in effect, that in trying to determine the best investment for a non-profit organization, rules that base their recommendations on tax bracket are not likely to be successful. The second indicates that when dealing with clients nearing retirement age, more secure stocks should be considered before more speculative ones.

```
METARULE001
If  1) you are attempting to determine the best stock
       to invest in,
    2) the client's tax status is non-profit,
    3) there are rules which mention in their premise
       the income-tax bracket of the client,
then it is very likely (.9) that each of these rules
     is not going to be useful.

PREMISE
 ($AND(SAME OBJCT CURGOAL STOCK-NAME)
      (SAME OBJCT STATUS NON-PROFIT)
      (THEREARE OLRULES ($AND
            (MENTIONS FREEVAR PREMISE BRACKET)) SET1))
ACTION (CONCLUDE SET1 UTILITY NO .9)

METARULE002
If      1) the age of the client is greater than 60,
        2) there are rules which mention in their
           premise blue-chip risk,
        3) there are rules which mention in their
           premise speculative risk,
then    it is very likely (.8) that the former should
        be used before the latter.

PREMISE
($AND(GREATER OBJCT AGE 60)
     (THEREARE OLRULES ($AND
            (MENTIONS FREEVAR PREMISE BLUE-CHIP)) SET1)
     (THEREARE OLRULES ($AND
        (MENTIONS FREEVAR PREMISE SPECULATIVE)) SET2))
ACTION
    (CONCLUDE SET1 DOBEFORE SET2 .8)
```

Figure 11 - two meta-rules

It is important to note the character of the information conveyed by meta-rules. First, note that in both cases we have a rule which is making a conclusion about other rules. That is, where object level rules conclude about the stock market 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 their *validity.* That is, **METARULE001** does not indicate circumstances under which some of the object level rules are invalid (or even "very likely (.9)" 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., delete the meta-rule, and just add another premise clause to each of the relevant object level rules. But since their conclusions do concern utility, it does not make sense to distribute the knowledge.

Adding meta-rules to the system requires only a minor addition to the control structure described above. As before, the system retrieves the entire list of rules relevant to the current goal (call it L). But before attempting to invoke them, it first determines if there are any meta-rules relevant to that goal[4]. If so, these are invoked first. As a result of their action, 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.

{4.4.3} Meta-rules: guiding the use of the knowledge base

There are several points to note about this approach to encoding knowledge. 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 search terms, we are talking about the difference between "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 part of 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 Section {4.4.2} has been simplified in several respects for the sake of clarity. It discusses the augmented control structure, for example, in terms of two levels -- the object and meta-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 which can be used to reorder or prune L. But before invoking this, it checks for second order meta rules which 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.

Consider once again the issue of leverage, and recall the value of heuristics in guiding tree search. We can apply the same idea at this higher level, gaining considerable leverage 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.

The judgmental character of the rules offers several interesting capabilities. It makes it possible, for instance, to write rules which make different conclusions about the best strategy to use, and then rely on the underlying model of confirmation [16] 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 11 means that this tree has an interesting characteristic: at each node, when the system has to

choose a path, there may be information stored advising 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 which 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 which are goal-specific, explicit, and embedded in a representation which is the same as that of the object level knowledge. That fact that strategies are *goal-specific*, for instance, makes it possible to specify quite precise heuristics for a given goal, without imposing any overhead in the search for any other goals. That is, there may be a number of complex heuristics describing the best 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 ease of modification of established strategies. Consider, for instance, alternative means of achieving the sort of partial ordering specified by the second meta-rule in Figure 11. 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 [1]. That is, rather than dealing with a linear list L of relevant rules, those rules would be put on an agenda. Partial ordering could be accomplished simply by setting the priority for some rules higher than that of others: rules in subset A, for instance, might get priority 6 while those in subset B were 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 higher priority than 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 each may be? Consider also what happens if, before we get a chance to invoke any of the rules in A, an event occurs which 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 are 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 searching through a program for all the places 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's 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 of a correspondingly different type as well.

{4.4.4} Meta-rules: broader implications

There are a number of interesting generalizations of the basic scheme presented above, two of which we touch on briefly here. First, while we have been examining the idea of strategies in the context of the depth-first search used by the performance program, the concept is in fact more widely applicable and can be used with a range of control structures. Second, meta-rules effect their selection of the relevant object level rules by what we have termed *content-directed invocation*, an approach which offers advantages over previous knowledge source invocation techniques.

Applications to other control structures

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 that case meta-rules offer a way of controlling the depth and breadth of the implications drawn from any new fact or conclusion. Pursing this further, we can imagine making the decision to use a data- or a goal-directed process itself 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.

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 11 refer to the object level rules by *describing* them, and effect this description by direct examination of content. For instance, METARULE001 refers to *rules which mention in their premise the income tax bracket of the client*, a description, rather than an equivalent list of rule names. The set of object level rules which 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 (in this case the presence of the attribute BRACKET), 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 (e.g., if METARULE001 had been written to indicate "`it is very likely (.9) that RULE124, RULE065, RULE210, and RULE113 are not going to be useful`"). If such lists were used, then tasks like editing or adding an object-level rule to the system would require extensive amounts of bookkeeping. After an object level rule has 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. By using 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.

Additional advantages of this technique are discussed in more detail in [2] and [5].

{5} Implications

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 [2] 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., [18], [13]) have attempted to assign meaning to the code of some standard programming language. Rather than take on this sizable task, we have used the task specific languages to make the problem far easier. Instead of attempting to assign semantics to ordinary code, a "meaning" is assigned to each of the primitives in the high level language, and represented 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 [2] for details).

This convenient shortcut also implies a number of limitations. Most important, 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 [6] and [17] for examples); they are viewed as data structures, and dissected and abstracted to form the rule models; they are dissected and examined in order to produce explanations (see [2]); they are constructed during knowledge acquisition; and finally they are reasoned about by the meta rules.

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 any representation with simple enough internal structure and of mangable 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 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 important, what knowledge was required to solve it.

But what if we reversed this trend, and instead view 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.

{6} 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.

Notes

(1) TEIRESIAS was developed in the context of the MYCIN system [17,6], which deals with infectious disease diagnosis and therapy. The domain has been changed to keep the discussion phrased in terms familiar to a wide range of readers, and to emphasize that neither the problems attacked nor the solutions suggested are restricted to a particular domain of application. The dialogs shown are real examples of TEIRESIAS in action, with a few word substitutions: e.g, *primary bacteremia* became *Georgia Pacific*, *infection* became *investment*, etc.

(2) Both of these are constructed via simple statistical thresholding operations.

(3) This was suggested by the perspective taken in work on SMALLTALK [9] and actors [10].

(4) 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 paper. The issues of organizing and indexing meta-rules are covered in more detail in [2] and [5].

References

[1] Bobrow D, Winograd T, An Overview of KRL, *Cognitive Science*, vol 1, pp 3-47, Jan 1977.

[2] Davis R, Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases, Stanford HPP Memo 76-7, July 1976.

[3] Davis R, Knowledge about representations as a basis for system construction and maintenance, to appear in *Pattern-Directed Inference Systems*, Academic Press, (in press).

[4] Davis R, Interactive transfer of expertise, to appear in *Proc 5th IJCAI*, Aug 1977.

[5] Davis R, Generalized procedure calling and content-directed invocation, to appear in *Proc AI/PL Conference*, Aug 1977.

[6] Davis R, Buchanan B G, Shortliffe E H, Production rules as a representation for a knowledge-based consultation system, *Artificial Intelligence*, **8**:15-45, Spring 1977.

[7] Feigenbaum E A, et. al., On generality and problem solving, in *MI6*, pp 165-190, Edinburgh University Press, 1971.

[8] Fennell R D, Multiprocess software architecture for AI problem solving, PhD Thesis, Computer Science Department, CMU, May 1975.

[9] Goldberg A, Kay A, Smalltalk-72 User's Manual, Learning Research Group, Xerox PARC, 1976.

[10] Hewitt C, A universal modular actor formalism for AI, *Proc 3rd IJCAI*, pp 235-245.

[11] Lesser V R, Fennell R D, Erman L D, Reddy D R, Organization of the HEARSAY II speech understanding system, *IEEE Transactions*, **ASSP-23**, February 1975, pp 11-23.

[12] Mathlab Group, MACSYMA reference manual, 1974, MIT.

[13] Manna Z, Correctness of programs, *Journal of Computer Systems Sciences*, May 1969.

[14] Samuel A L, Some studies in machine learning using the game of checkers II - recent progress, *IBM Jnl Res and Devel*, 11:601-617.

[15] Schank R C, Abelson R P, Scripts, plans and knowledge, *Proc 4th IJCAI*, pp 151-157.

[16] Shortliffe E H, Buchanan B G, A model of inexact reasoning in medicine, *Math Biosci* 23 (1975) pp 351-379.

[17] Shortliffe E H, *MYCIN: Computer-based Medical Consultations*, American Elsevier, 1976.

[18] Waldinger R, Levitt K N, Reasoning about programs, *Artificial Intelligence*, 5 (Fall 1974) pp 235-316.

[19] Winograd T, *Understanding Natural Language*, Academic Press, 1972.

[20] Winston P, Learning structural descriptions from examples, MIT TR-76, Sept 70.