Report 81-21
Stanford -- KSL

Scientific DataLink

Meta-Cognition: Reasoning about Knowledge.
Douglas B. Lenat, Randall Davis, Jon Doyle,
Michael R. Genesereth, Ira P. Goldstein,
Howard Schrobe, Dec 1981

card 1 of 1

BLANK PAGE

Meta-Cognition: Reasoning about Knowledge

Douglas Lenat, Randall Davis, Jon Doyle, Michael Genesereth, Ira Goldstein, Howard Schrobe

1. Current problems in building expert systems

The process of designing, developing, and using expert systems has associated with it a number of basic problems:

(i) Rule location: As the number of rules in the expert system grows, it becomes less practical to test the conditions of every rule, each time the situation changes slightly. There must be some way to efficiently find the relevant rules. In the context of diagnosing an unknown oil spill seen on a stream, rules about bridge bidding should not even be examined for potential relevance.

(ii) Conflict resolution: Often, several rules will be relevant (have all their conditions satisfied) in a given situation. Which rule should be chosen? Should several be obeyed immediately, or should just one be chosen for execution?

(iii) Knowledge Base Revision: As time goes by, new information may exist and some old information may be outdated. This typically occurs gradually, but also may occur as a result of a sudden emergency (e.g., "We just ran out of lime; what should we use instead?")

(iv) Explanation: The end users of the expert systems have a healthy disrespect for machines simply telling them what to do. No doctor wants to rely on an expert system's diagnosis or prescription, without first asking for an explanation. The expert system's response to such an inquiry may initially be a trace of its line of reasoning. But the careful expert will often be unsatisfied with this, and will want to probe deeper in a few areas critical to the present case. He or she may want to know the justifications for individual rules, or even for the global diagnosis strategies the system is relying on.

(v) System Explanation: Sometimes the user's questions will be so deep that they are actually asking about the control structure and organization of the expert system itself, its assumptions about the task, etc. These are questions that typically require both a domain expert and the system builder to answer.

(vi) System Reorganization: An even more extreme kind of revision than that in (iii) above occurs when the system has been in use for a while. Based on that experience, it may need to be reconfigured, possibly substantially rewritten. Current approaches to solving these problems involve providing the expert system with additional types of information, and therefore additional capabilities.

For problems (i), rule location, the knowledge we have to add is more or less strategic. The system needs strategies about how to quickly focus in on the relevant

categories of rules. An example of this might be a piece of knowledge that said

"If you are currently seeking rules about X, Then look first at rules which mention X somewhere inside them; look next at rules that mention some generalization of X."

Thus, when an unknown oil spill is noticed, the first rules the expert system considers are those that explicitly mention oil spills. If none of these are found to be relevant, the system might gather rules about spills in general. Only if none of these rules is found to be relevant would rules about, say, bridge bidding finally be inspected.

The way to treat problem (ii) is quite similar: one must add strategies for resolving conflicts, when more than one is relevant at the same time. An example is "Choose randomly". A real time-saver is "Choose the first one you find". A less mindless strategy might be "Choose the rule entered by the most experienced expert", or "Choose the rule that takes the least time and energy to carry out".

A different type of knowledge must be added to the system to overcome problem (iii). The justifications, origins, empirical experience, and other descriptive information about the rules is needed. Thus, when a rule says to apply lime to the spill, that rule also has knowledge about why lime is being recommended. If, in an emergency, no lime is available, the system will be able to choose a substance that has needed properties (sometimes just pH matters, sometimes formation of precipitates matters, etc.) If a new, cheap base is marketed one day, the system will possess the necessary knowledge to go through its rules and make changes where the new substance is preferable to the old choice.

To explain its reasoning to users, problem (iv) above, the system needs a similar kind of knowledge about its own rules: their justifications, origins, and how they have behaved empirically. Thus, when an expert feels uneasy about some rule, he can ask who said so, or why it works, or how well it has performed in practice.

This type of descriptive knowledge will not in general be adequate to answer the deep system questions raised in (v) above. For that, we need to add knowledge about the design of the expert system itself. One such body of knowledge deals with the assumed model of what the task is and how it is to be carried out. Next comes general information about building expert systems: the control structures that are available, the ways of representing knowledge, and the advantages and disadvantages of each choice. From these two bodies of knowledge, the system can answer questions like "Why was such a slow control structure chosen? Why can't the system 'back up'? Why was each type of oil represented just as a vector of numbers?"

That same knowledge -- the model of the task plus some general facts about building expert systems -- is precisely what we need to write the expert system in the first place. When our skills at automatic programming improve, this knowledge could be used to tackle problem (vi), dynamically reconfiguring, reorganizing, and rewriting the system. An example of some of this knowledge is "If the search space is small, Then it's fine to just search exhaustively".

2. Meta-Cognition

The types of processing just described are called *meta-cognition*, and the knowledge we must add is called *meta-knowledge*. The prefix "meta-" means "knowledge about --", so meta-knowledge means knowledge about knowledge.

To tackle problems (i) and (ii), *strategic meta-knowledge* must be supplied to the system: ways to locate and choose among rules. Often these strategies are themselves phrased as IF-THEN rules, and in that case we refer to them as meta-rules. Strategic meta-knowledge will, in general, improve the performance of the program because it guides and constrains the search for a solution.

To tackle problems (iii) and (iv), *descriptive meta-knowledge* is supplied: the justification for each rule, its origin, records of how it has performed in actual use, etc. Descriptive meta-knowledge makes it easier for a human -- or even for the program itself -- to make changes to the program; this increased flexibility is often worth the added effort of supplying the descriptive information initially.

Finally, problems (v) and (vi) require the use of *systemic meta-knowledge*: the model of the system's task, and information about building expert systems, including enough to explain each design decision in the current system. As programs grow ever larger and more complex, it becomes increasingly difficult for human beings to stay "on top" of everything that is happening internally. The program itself must assume more and more of the burden of understanding its own behavior, documenting and justifying itself, and even modifying itself. To illustrate some ways of meeting this need, this chapter traces through the evolution of a program as we add to it a variety of forms of competence:

performance: Can it solve the problem?
explanation: Can it explain how it solved the problem?
learning: Can it improve at solving the problem by learning more about the domain?
self-description: Can it introspect on its own representations and control structure?
self-modification: Can it alter its own design in response to experiences it has?

This sequence of competences closely parallels the real concerns that have developed historically. Initially the only concern was performance. Later, the answers provided by some programs became too difficult to "check" immediately, and it became necessary for the programs to supply an English-like rationale or justification for their conclusions. One of the bottlenecks in building expert systems is knowledge acquisition, and even partially automating that process -- i.e., learning -- is becoming an important ability for a system to have.

Most long-lived programs are constantly being updated and changed. Documentation and comments help immeasurably in this process, and one step further along these lines is to give the program itself the ability to introspect -- and explain -- its own data structures, representations for knowledge, control structure, etc. This is a much deeper level of introspection and self-explaining than the performance-rationale capability mentioned previously.

An even more dramatic ability would be that of self-modification: a program being able to redesign itself, its data structures, its algorithms, in response to changing

demands. The program itself might monitor its runtime environment and perceive these new demands itself, or a human might explicitly tell it some change in its specifications. This capability -- self-reconfiguring -- is aimed at capturing the competence of the expert system designer.

In part this also reflects a broadened perspective on what constitutes competence. An expert is more than someone who merely solves a problem. He is capable as well of explaining, learning, reorganizing, reformulating, and dynamically rating the progress he's making. All of these forms of knowledge we consider meta-knowledge, and we will explore them in the sections that follow.

Looking over the knowledge provided by the experts on the oil spill task, we see that much of it deals with "facts", such as the various agencies which must legally be notified in the event of a large oil spill. The remaining knowledge is in the form of "heuristics", rules of thumb which help guide you in performing actions, such as how rapidly each agency typically responds once it's notified. Some of this heuristic knowledge refers specifically to other knowledge, e.g., just how important is it to notify each agency. That last category, knowledge about knowledge, is not strictly necessary to perform the oil spill task. But the flexibility and ultimate ultilty of the system may depend on how much of this meta-knowledge is incorporated.

What is the "source" of meta-knowledge? Paradoxically, much of it derives from an incomplete understanding of the phenomena. Until a task has been completely formalized -- and often after as well -- experts will not discuss it in formal terms. First they provide a base of facts, theorems, equations, categories, lab objects and operations, etc. This comprises a "zero-th order" theory of their task domain. It contains the factual knowledge present in texts on the subject. Often, the experts claim that this knowledge is complete, and only after a program emboying it fails to perform adequately do they volunteer additional knowledge. This second sort of knowledge is rules of thumb, inconsistent advice, inexact judgmental criteria, etc., and it forms a "first-order correction" to the factual zero-th order theory. But often, even this knowledge does not produce results as well as the experts. Continued pressure on the experts will yield a third type of knowledge, the kind we are calling meta-knowledge: "Well, this last piece of advice really doesn't work for paint spills", or "Chemical analysis isn't totally reliable". This is a "second-order correction" to the previous system knowledge.

This scenario is not always followed precisely, but it indicates the wide variety of information offered by an expert when describing a domain and the problem solving techniques he uses. Directly or (more often) indirectly, almost all expert systems contain meta-knowledge of the sort illustrated above.

If it's going to be present (and it always seems to be), we should deal with meta-knowledge directly. That is, we ought to recognize it and represent it explicitly in a language that is high level, economical in expression, and has the appropriate primitives. The program will function better and will be easier to build and improve as a result. Examples given throughout the rest of this chapter illustrate this point.

The spill amelioration problem domain has already been discussed in detail elswhere

in this book. For the remainder of this chapter, let's presume we have before us an archetypical program performing that task. Its architecture will start out as simple as possible; say it is a rule-based[1] system, with rules being cycled through randomly; anytime the current rule's "conditions" part is satisfied, it will fire off (execute) its "actions" part.

Suppose the following are two of the rules in the system:

```
R1:     if the spill is sulphuric acid, then use an anion-exchanger.
R2:     if the spill is sulphuric acid, then use acetic acid.
```

Once a spill has been identified as sulphuric acid, each of these rules will have their if-parts (conditions) satisfied. Which of them will be followed? Since the control structure is random selection, there is no bias, and no way of knowing which rule -- and which remedy -- will be selected. But in "real life" there must be some basis for choosing among them, or the expert probably wouldn't have two different methods to react to the same situation. How does he communicate that extra information? It is meta-knowledge, and the next three sections deal respectively with adding strategic, descriptive, and systemic meta-knowledge to our tiny system. It is important to note that most expert systems today have little or no meta-knowledge in them. This chapter is sketching long-term research directions and aspirations, and only some of what we anticipate will turn up as usable "knowledge engineering products" in the coming decade. Despite its speculative nature, the incorporation of metacognition into expert systems will probably play too important a role to ignore in this book.

3. Strategic meta-knowledge

Besides R1 and R2, suppose we now add three new rules to our system:

```
R3:  Use rules that employ cheap materials before those that employ more
            expensive materials.
R4:  Use less hazardous methods before more hazardous methods.
R5:  Use rules entered by an expert before rules entered by a novice.
```

Note that these new "meta-rules" require that the system be able to reason about the items mentioned in rules R1 and R2. In particular, there must be facts in the system that acetic acid is cheap and anion-exchanging is generally more expensive; that acetic acid may be more dangerous to work with than anion-exchangers; that R1 was typed in by Dr. Johnson (Oak Ridge's spill expert) and R2 by Bob Smith, a sophomore spending the summer at Oak Ridge.

---

[1] We are using a rule-based system as a focus for the discussion, but none of the material here is constrained to a rule representation. All of the discussion is oriented to issues of kinds of knowledge and the appropriate organization for that knowledge; we do not deal with issues that are particular to a specific representation.

This "content-reference" is easily resolved automatically, and it is clear that R3 and R5 would cause R2 (acetic acid) to be considered before R1, while R4 would recommend that R2 be considered after R1.

So which rule -- R1 or R2 -- will be executed first? It comes down to: which of R3, R4, R5 will be executed first? If there is a complete separation of rules from meta-rules, and a separate interpreter for meta-rules, then the question becomes dependent upon that particular meta-rule interpreter. If it's pure random selection, then there is a 2/3 chance of R2 being tried before R1 (since two rules favor R2 over R1), and therefore that acetic acid will be the method of choice.

But let's take a look at R3, R4, and R5 again. They all give good advice -- and quite general advice. Why not permit them to affect the meta-rule interpreter as well as the rule interpreter? Consider R5, first. It says to prefer experts' rules over novices' rules. If R4 was typed in by Dr. Johnson, and R3 by Bob Smith, the advice says to prefer R4 to R3. That will result in anion-exchanging ultimately being chosen. Now consider R4: prefer less hazardous methods. Suppose R3 is known to sometimes cause catastrophes, R5 is less risky, and R4 is safe but sure. Then even R4 prefers R4 to the other two rules. Finally, consider R3: pinch pennies. Suppose that estimating the cost of various approaches is itself difficult and costly to do, estimating the risks involved is less costly, but finding out who typed in a rule is impossible since it was never recorded. R3 then prefers R4 to the other two rules. In short, all of R3, R4, and R5 prefer R4 as the "meta-rule of choice" in this problem, and therefore R1 -- and hence anion-exchanging -- is definitely selected. Of course in general the meta-rules will not all prefer the very same meta-rule; some kind of meta-rule interpreter must be specified to resolve such conflicts. The key idea of this paragraph was merely to show that sometimes it is better not to make a distinction between knowledge and meta-knowledge, that sometimes rules can quite profitably apply to each other and to themselves.

A related point is that one person's meta-knowledge is another's knowledge. Consider the following:

```
K1:  The backtracking approach is usually tried first if the source is
unknown.
```

Is it knowledge, or is it meta-knowledge? If your task is to ameliorate the spill, then planning how to do it is at a meta-level to you, and you'd call K1 meta-knowledge. Knowledge, to you, are pieces of information about how to backtrack, what manhole is connected to what pipe, etc. But if your task is *planning* to clean up the spill, then K1 is just a piece of knowledge to you.

A final related issue concerns the worry about having to provide meta-meta-knowledge, and then meta-meta-meta... ad infinitum. This is not the brink of disaster for several reasons: First, the number of meaningful rules at each meta-... level decreases sharply. By the time you've supplied two or three meta's, the only rule you have left is "Maximize utility". Second, as we saw above, there may not be any need to disassociate rules from meta-rules, and of course this means there is no need for them not to also serve as meta-meta-rules, etc., whenever they can be meaningfully applied. Third, one can simply omit any explicit rules from one level onward; the system will still behave better than if no lower-level rules had been provided. E.g.,

one can arbitrarily decide that meta-rules will be present, but no meta-meta-rules. The system still will reason more deeply than any system without any meta-rules.

Work to date

Strategic meta-knowledge was historically one of the first forms to be recognized, though it was generally encoded indirectly. By "indirectly", we mean that various problem domains forced system builders to contend with it, but they often did so with mechanisms that were added on to their system designs as afterthoughts. The idea has shown up in many places: (i) Conflict resolution in production systems; (ii) Advice lists in Planner; (iii) Meta-rules in Teiresias; (iv) Choice rules in NASL; (v) Layered planning in Molgen. This description of work to date seems to brief, and the reader may get the impression that not much has been done. That is correct.

4. Descriptive meta-knowledge

Suppose our system has these three rules in it:

```
R6:   if spill is sulphuric acid, then use lime.
R7:   if spill is acetic acid, then use lime.
R8:   if spill is hydrochloric acid, then use lime.
```

Now suppose lime runs out and lye is suggested as a replacement. Presumably we know many additional things about lye, e.g., the compound formed by lye and sulphuric acid is soluble. Can we simply go through rules R6-R8 and replace lime by lye? If we do that, some of them still work and some of them become completely useless (or even worse than useless). What is wrong?

Let's take a deeper look into the rationale behind those rules. Suppose we had taken the trouble to supply some descriptive meta-knowledge about each one: whenever a rule is typed in, the expert is asked to also specify its justification:

```
R6: if spill is sulphuric acid, then use lime.
        Justification: lime neutralizes acid and the compound that forms
                       is insoluble and hence  will precipitate out.
R7: if spill is acetic acid, then use lime.
        Justification: lime neutralizes the acid.
R8:  if spill is hydrochloric acid, then use lime.
        Justification: lime neutralizes the acid.
```

The suggestion about using lye instead of lime also had a justification:

```
S1: if you run out of lime, then consider using lye instead.
        Justification: lye can neutralize acids (just as lime does)
```

What we really want to do is go through R6-8 and substitute lye for lime only in those rules which use lime solely to neutralize pH -- i.e., not in rule R6. The point is that substitutions work only when you know why the original compound was being employed in the first place.

This new type of meta-knowledge -- justifications - is quike unlike the strategic meta-knowledge described in Section 3. There, the meta-knowledge was effective, executable, runnable advice. It constrained and guided a performer as he searched for a solution. Here the meta-knowledge is non-effective, descriptive, explanatory. In both cases, though, it has proven itself useful.

Indeed, if you examine Section 3's strategic meta-knowledge carefully, you'll see that it requires that a great amount of descriptive meta-knowledge be available. The strategic meta-rules demand information about the system's rules. For instance, R5 demands to know who typed in each rule in the system. R5 can't even function unless that type of record-keeping has been done.

What kinds of descriptive meta-knowledge are typically called for by the strategic rules? Often, there is a need to know just what makes you believe something, what the justification for a rule is. Is it by definition? by appeal to an established authority? by deduction? by induction? by observation? Except for those rare cases of ironclad justifications, most of Polya's rules [1954] for the use of evidence in plausible (inexact) reasoning are quite appropriate here.

Even given the truth of a rule, the rule itself may have reservations about its conclusion. The expert might qualify the rule's actions-part by saying "Then it is likely that...", perhaps even assigning a numeric certainty factor or likelihood ratio. Besides the average or mean "truth" of a rule, it may be worth recording its variance; we refer to this as the reliability of the rule. For instance, in a desparate emergency, one might employ methods with only mediocre average justifications, if their variance were high enough to give some chance for success.

A related measure is the downside risk -- just how bad is it if the rule gives incorrect advice? For example, "If we fully automate our retaliatory strike launchings, Then global peace is assured" might be right most of the time, but if it's ever wrong...

Omnipresent in most meta-cognitive systems is an awareness of resources -- time, space, money, effort, etc. For example, in the spill task, one crucial bit of descriptive meta-knowledge is the fact that it takes several days to get the results of a general chemical analysis back from the lab. In some rule-based systems, the methods (and even the rules and meta-rules) are monitored, and records of their average running time, percentage record of success, etc. are gradually accumulated. This empirically-obtained resource data often turns out to be more accurate than an expert's *a priori* intuitive estimates.

Yet another useful type of descriptive meta-knowledge is some notion of how accurate the result obtained is, from a given method. You may get five digits back, but how many are significant? For example, in the oil spill task, weathering of a sample will reduce the trust you should put in the results of its chemical analysis.

So far, all the descriptive meta-knowledge has dealt with the content of the rules and methods in the system; it is possible to deal with the form of the rules (and other knowledge-carrying structures) as well:

```
FR1:  Use lye.
```

A rule such as FR1, having no conditions, should be noticed, and brought to the attention of the person who typed it in for confirmation. Probably, the user meant that the context of this rule (its condition) was assumed to be the same as the one(s) he's just been typing in. Actually, that's not a bad piece of strategic advice, and we call it FR2 and add it to our system:

```
FR2:  If a rule is typed in with no conditions, then ask the typist if he
meant it to have the same conditions as the most recently typed in rule.
```

Another "bug" that occurs frequently is for a rule to have conditions that never can be satisfied:

```
FR3:  If acid must be neutralized, and lye is available,
         and no basic material is available,
      then use lye.
```

In the case of FR3, above, the user really meant to type "...and no other basic material is available,...", but since he left out the word "other" the rule as stated will never fire. This might possibly be detected, either deductively by proving that FR3's clauses are not simultaneously satisfiable, or empirically by recording the behavior of FR3, and employing a rule such as FR4:

```
FR4:  If, over the course of many runs, a rule never fires, then ask the
expert if the condition is for some (semantic) reason unsatisfiable, and
request him to rephrase it.
```

Both FR1 and FR3 exhibit rules whose form has a bug in it. In the first case -- no condition -- the error is syntactic. In the second case -- an unsatisfiable condition -- the error is semantic. In each case, descriptive meta-knowledge can be used to detect and correct such bugs. In the first case, the descriptive meta-knowlege is that all rules should have both conditions and actions. In the second case, the descriptive meta-knowledge is that all rules should fire at least sometimes. These pieces of descriptive meta-knowledge are therefore quite useful.

Descriptive meta-knowledge is useful in knowledge acquisition and knowledge-base evolution. We saw at the beginning of this section how changes in the knowledge base are less prone to errors if each rule -- and each change -- is supplied with an explicit justification. We saw above how descriptive meta-rules can monitor and analyze the form of newly typed-in rules, sometimes spotting a common bug and immediately doublechecking it with the human.

Knowledge acquisition and knowledge base evolution have also been aided by the use of descriptive meta-knowledge in the form of knowledge structure syntax. In systems whose knowledge-carriers (rules, frames, etc.) have a great deal of structure to them, knowledge acquisition becomes a process of filling in the schemata; the "dialogue" between system and expert is more like text-editing or prompt-answering than like unguided type-in. A simple example of this occurs after we've typed in rule R7, and want to type in R8. The structures, justifications, etc. are so similar, that it is easiest to simply edit a copy of R7, and replace the word "acetic" by "hydrochloric".

Such schemata only help describe the *syntax* of the knowledge base. Descriptive meta-knowledge on interrelations which should exist among the knowledge (either statically, such as inverse links, or dynamically, such as relative running times) can be used to provide checks on the *semantics* of newly acquired knowledge.

A more sophisticated example involves the use of information about the representation of knowledge in the system, some descriptive meta-knowledge about a relation between two of the "slots" of a frame-structured representation:

```
FR5: If a rule has an "If-Potentially-Relevant" slot filled with some
predicate PR, and an "If-Truly-Relevant" slot filled with predicate TR,
     then TR should imply PR, and PR should be much faster to execute than TR.
```

The idea here is that a rule can have two parts to its conditions: a fast but uncertain pre-pre-conditions check (If-Potentially-Relevant), and a slow, sure, detailed condition (If-Truly-Relevant). For instance, FR3 might have been split into three parts:

```
FR6:   If-Potentially-Relevant: Acid must be neutralized
       If-Truly-Relevant: Acid must be neutralized, lye is available, and no
            basic material is available
       Then: Apply lye to the spill to neutralize the acid
```

FR5 states explicitly how the semantics of the two kinds of If-parts interrelate. It is a piece of descriptive meta-knowledge. FR5 declares that the If-Potentially-Relevant slot must be implied by -- and faster than -- the If-Truly-Relevant slot, and that is certainly the case in FR6. Coupled with, say, some statistical runtime data, it might be possible to verify that indeed, for some other rule, R99, its "fast" condition was taking longer to test than its "slow but sure" one. In that case, some strategic meta-knowledge might cause the system to do something (e.g., log a message to that effect, interrupt the user and warn him, delete the If-Potentially-Relevant slot of R99 completely, etc.)

We have established five uses for descriptive meta-knowledge:

(i) To provide explanations of the goals underlying the program's actions. Traditionally, the question WHY? is answered via a replay of the inference performed. Descriptive meta-knowledge can provide a deeper answer.
(ii) To help the system's strategic meta-knowledge answer questions such as "Did an expert or a novice write R2?" "Does R1 or R2 run faster?"
(iii) To help detect syntactic and semantic bugs in the form of recently-entered rules, thereby aiding knowledge base expansion.
(iv) To facilitate the entry of new knowledge -- terms, facts, heuristics -- by providing a structured corpus of already-existing knowledge. Instead of typing in the new knowledge from scratch, the expert can simply browse, point to similar knowledge, copy it, and then edit it to produce the new knowledge, and
(v) To explain the semantics of the relations between slots.

Work to date

Certainty factors are added to rules by the physician-experts who build the Mycin knowledge base [Shortliffe 76]. Similarly, the geologist-experts who add rules to Prospector [Duda 79] provide a likelihood ratio for each rule. Eurisko [Lenat 82] keeps track of the average running times, space consumed, success and failure rate, etc. of its rules, and uses these statistics to affect its choice of which rule to try in new situations. A few expert systems pay attention to the form of the rules. For instance, Eurisko will complain if the user tries to type in a rule with no conditions (a piece of syntactic meta-knowledge complains). If the user attempts to type in a rule whose Pre-pre-conditions take longer to run than its pre-conditions, a piece of semantic meta-knowledge complains, after a great deal of empirical run-time statistics have been accumulated. Teiresias [Davis&Lenat 81] may occasionally interject a comment like "Excuse me, but all rules mentioning x and y have also mentioned patient age as a relevant parameter; are you sure you don't want to add some clause about patient age?"

A number of programs have worked within the model of using descriptive meta-knowledge to provide explanations of the program's goals. Mycin[Shortliffe 76] was one of the first. A slight improvement is seen in [Swartout 77], where, instead of simply replaying the exact rule used, the system determined what generic principle the rule was an instance of, and printed that instead. This produced somewhat more general and often somewhat more comprehensible explanations. While it was still following the "retrace the performance" model of explanation, the restructuring of the program as a collection of domain principles helped to make that performance more understandable.

5. Systemic meta-knowledge

Loosely speaking, systemic meta-knowledge is that knowledge about global system design that exists in the head of the knowledge engineer. What is it that a good system designer knows and puts to work in coming up with, or even simply selecting from among such architectures as Macsyma, Hearsay-II, Mycin, Harpy, AM, Dendral, etc.? Why prefer a blackboard model, or an agenda, or a pure production system? Will resolution work? Do you need metarules? Can you afford to do exhaustive backward-chaining? See, e.g., [Feigenbaum 77] and [Stefik 82].

Such decisions call for knowledge about the consistency and the completeness and magnitude of the available knowledge, the size and shape of the search space, the required efficiency of the system, and on similar parameters of systemic -- global -- scale. For a detailed discussion of these options, see [Stefik 82].

Let's add some systemic meta-knowledge to our sample spill system to illustrate three uses of systemic knowledge: designing an expert system, self-redesigning, and design-explanation. Here are two new systemic meta-rules for the system:

```
R9:   if the search space is moderately small,
      then exhaustive search is feasible
R10:  if the rule set is rarely modified,
```

```
then it's worth compiling it initially into a decision tree
```

Thanks to facts such as "There are 33 standard oil types... and 1000 different locations...", the original designer could estimate the size of the search space involved, and -- employing R9 -- opt for an exhaustive backward chaining search.

Our original design presumed that there was an interpreter crawling over the rule set during execution, inspecting the conditions of many (if not all) rules before carrying out any action. But once our program has been debugged, tested, and shipped out into the field, the rules will rarely (if at all) change. R10 might then fire, and redesign (or at least recommend that somebody redesign) the basic control structure of the system, so that it followed a decision tree to obtain an answer.

If the program begins to be used on larger and larger problems, R9 may eventually complain that it is being violated, and that an exhaustive search may no longer be such a great idea -- again, we see a rule suggesting a redesign.

Suppose a user is employing the program, and asks why a particularly unlikely rule was even considered. The system replies that it was executing an exhaustive search, and the user asks Why? again to that. This time, R9 is printed out to answer the query. This is a significant change from answering "why?" questions by typing out the control stack (higher level goals); after you get to the top of that stack, the system can begin to interpret "why?" to mean "why was the system designed like that?"

Thus we've seen the same rule -- R9 -- used in three distinct roles: to aid in designing a system, to aid in knowing when and how to redesign a system, and to answer "why?" questions of a fundamental nature. Let's look at each role in more detail.

The most obvious use for systemic meta-knowledge is to aid in designing and constructing a system. At present, the process of building an expert system is long and painful, due to the difficulties in extracting knowledge from human experts. If we actually had a large codified corpus of systemic meta-knowledge, this process might be ameliorated. It may be many decades before we can exhibit a system which can actually build expert systems on its own, but long before then we might have one which can usefully advise a human who is attempting to design and build one.

A second use for systemic knowledge is system re-construction and self-modification in general. Can the system reconfigure itself in response to experience, to what it may have learned about the domain? Here are two rules which support that type of operation:

```
R11:  If a piece of code is called frequently, then it's worth optimizing it.
R12:  If (almost) every frame has many entries on its s slot,
      then try to replace s by a cluster of more specialized slots.
```

Our original design had rules possessing only "conditions" and "actions" slots. If the majority of rules have large conditions (many clauses), then R12 might fire and suggest dividing that slot up into a constellation of new, more specialized conditions slots: If-potentially-relevant, If-there-are-enough-resources, If-truly-relevant, etc.

Instead of each rule having a dozen clauses on its conditions slot, it might then have three or four clauses on each of the new, specialized conditions slots. Note that the "If-truly-relevant" slot need contain only those conditions which are not also present on the other "If-" slots.

The wisdom of such a partitioning rests with the Justification of R12, which might cite evidence as solid as "simplifies the entry of new rules", or as dubious as "appeals to the aesthetic tastes of the end users".

The third use for systemic knowledge is to provide a new form of explanation. Consider a program that's in the midst of the square-root calculation. If we interrupt it and ask what it's doing, the answer might be that it was dividing two numbers. If we ask why, the answer is "in order to get the square-root, via Newton's method." A second "why" might mean either, why are you computing a square-root, or why via Newton's method. In the second case, we go outside the goal/supergoal framework of typical explanation systems to explain why one of a number of design choices was made (in this case, the choice of algorithm). Recalling the end of the previous paragraph, the system might answer a "why?" question about a specialized new slot (the handiwork of R12) by citing R12 itself, and if another "why?" question follows, by citing the Justification of R12.

Work to Date

An early thesis by Jim Low [76] dealt with monitoring the manner in which each user accessed a file system, and then dynamically changing the data structures to improve efficiency. More recent work along similar "automatic programming" lines is [Barstow 79][Kant 79], the latter performing a Knuthian analysis of the various options. At MIT, the Macsyma advisor project [Martin and Fateman 71] and the programmer's apprentice project [Rich and Schrobe 76] embody much of what we have labelled systemic meta-knowledge. AM[Lenat 77] had rules of this form, such as "If a piece of code is executed twice, Then compile it", and Eurisko[Lenat 82] has employed rules akin to R12. The modern truth maintenance systems gain much power from explicit systemic meta-knowledge; see [Doyle 80]. As with earlier "Work to Date" sections, this one is brief to give the reader the correct impression that very little has yet been done in this area.

6. Learning, Self-describing and Self-modifying systems

The builder of an expert system is faced with a tradeoff: he wants to develop his system quickly and painlessly, yet he wants it to run as efficiently as possible. These two properties -- expressivity and efficiency -- are usually incompatible, and indeed meta-knowledge is heavily biassed to the former extreme. For that reason, many system builders decide they need efficiency, and by and large omit meta-knowledge from their system. But in this section we show how, by a kind of learning, programs with meta-knowledge can *recapture* some of this lost efficiency.

In artificial intelligence, it is often necessary to develop programs in an expressive

programming environment, such as Interlisp or MacLisp, without regard for runtime efficiency. Trying to build an efficient version from scratch leads to failure. So the experimenter concentrates on building a working experimental vehicle. Later, once it is running satisfactorily, it can be recast it in a more efficient form. In the Dendral and Mycin cases, this eventually led to minicomputer implementations, each requiring man-years of effort beyond the point where the programs ran satisfactorily in Interlisp.

Rather than having to manually translate the program into an efficient form, it may be possible to use techniques such as caching, learning, self-monitoring, self-modification, and intelligent compiling to automatically carry out that transformation.

Highly expressive programs employ techniques such as: reliance upon a very high level language (e.g., EMYCIN, ROSIE), planning and reasoning at multiple levels of abstraction, inference performed via pattern-directed knowledge sources (e.g., heuristic rules), and minimal, nonredundant representations (e.g., inheritance in a generalization hierarchy).

Very efficient programs have traditionally abandoned these techniques, though it appears this may not be inevitable. If an expressive program has a good model of itself and of programming, it may be possible for it to modify itself into a more efficient form. The basic source of power here is predictability: anticipate the ways in which you the program are most likely to be used, and reconfigure yourself to expedite such usage. Predicting program usage has long been exploited manually by human programmers, by analyzing static program descriptions.

We are advocating methods more dynamic than, e.g., Knuthian analysis. Rather than improve the static description of a program, we propose to have the program adapt to its runtime environment. Computer programs, no less than biological creatures, must perform in an environment: an externally imposed set of demands, pressures, opportunities, and regularities. Following [Lenat, Hayes-Roth, and Klahr 79], we use the term *cognitive economy* to mean the degree to which a program is adapted to its environment, the extent to which its internal knowledge structures and control processes accurately and efficiently reflect its environmental niche. In particular, four useful techniques for gaining cognitive economy are:

(i) **Self-monitoring**: sense, record, and analyze the way the program is being used
(ii) **Self-modification**: using dynamically acquired knowledge, redesign and recompile itself with a more appropriate representation, a faster algorithm, etc.
(iii) **Caching** of computed results: store the results of frequently-requested searches.
(iv) **Expectation-filtering**: use a model of the task domain to make predictions, which in turn set expectations. This can be used both to help ignore unsurprising data, and to watch for upcoming crucial data.

In various situations, at different times, both efficiency of compiled forms and accessability to declarative forms are intermittently needed. This argues for the economic benefit of maintaining simultaneously alternative and redundant representations for the same pieces of knowledge.

Tasks can (at least in principle) be specified anywhere from a pure *What* (desired i/o

behavior) to a pure *How* (precise algorithm). Feigenbaum [77] has observed that AI gradually pushes our ability from How toward What, e.g. by moving to a higher level language, a representation language, and ultimately an automatic programming system. Assuming then that "the user" wishes merely to specify What, an intelligent program must fill in the detailed code, the How. Traditional, static methods for program synthesis include formal program transformations [Burstall and Darlington 76][Balzer 80], and informal rules embodying knowledge of programming techniques and task-specific facts [Barstow 79][Green 74].

What might it mean for a program to modify itself dynamically? Suppose a hospital simulation program is specified, and on the basis of that user-supplied specification an event-based simulation seemed like the best design. After the program was used for a while, it became clear that most of the questions referred to the precise time of day, not to earlier and later events. A redesign, and reimplementation, in terms of a synchronous time-step simulation, seems called for. The attractiveness of having a system automatically notice its inappropriateness, diagnose a better design, and reconfigure itself accordingly, is apparent.

The simplest kind of adaptive abilities along this line would be to select a data structure, a representation, or an algorithm from a set of well-known choices, based on dynamic information about the current runtime usage of the program. For example, the following rules can decide to change a program's design based on such data:

```
IF one conjunct appears to be False more often than any other,
THEN reorder the conjuncts so that it comes first in the list.

IF each user is using the program only very briefly, and never more than once,
THEN throw away the code that spends time building up a model of the user.

IF a function is used frequently, and changed only occasionally,
THEN it is worth taking the time to compile it.

IF one part of the representation (e.g., one kind of slot) is being very
heavily used,
THEN consider replacing it by a cluster of similar, more specialized parts.
```

Now that we have illustrated the need for self-modification, and given a glimpse into techniques for achieving it, we return to the issue of self-monitoring. Since this really entails learning about the environment the program finds itself in, we are really talking about learning here. For decades, researchers have had ambitions to make their programs learn more about the problems they were attacking [McCarthy 68].

The simplest type of learning is by being told. Some task-specific item might be added along with the type-in of a problem, e.g. a strategic bit of meta-knowledge such as "Integration by parts might get you into an infinite loop on this problem", or a descriptive bit of meta-knowledge such as "This is known to be a very hard problem". Slightly less task-specific might be some strategic constraint governing the program's behavior over the course of the next several problems it's given, such as "Answers, even if uncertain, must be produced in real time or they will be useless to

me", or a descriptive long-range constraint such as "The current user for this whole session is one of the world's leading infectious disease specialists."

Learning by observing is the next step after learning by being told. An extreme case would be for the program to save a complete record of its entire runtime lifetime: all the inputs it received, outputs it made, function calls that were made, timing data for all of this, etc. From such a record, the program might deduce some useful information, e.g. "The compiler is practically never used, except in spurts". The space costs for such an archive, and the time costs for finding regularities in it, are both prohibitive. It is necessary to analyze in advance what features of the runtime world are worth recording or tabulating, and then gather only that data. Of course, the choice of what to measure can change with time. Some of the data worth collecting include: the frequency of calls on each function and rule, the most common arguments to each function, patterns in the sequence of function calls and rule firings (R93 always seems to precede R411), patterns in the outputs of functions and rules (EQUAL(x,y) usually returns NIL). Much research on this mode of learning needs to be done, and some recent AI efforts along these lines are [Buchanan and Mitchell 77], [Langley 79], [Michalski 77], and [Vere 77].

The most powerful type of learning is by using a model to make predictions. This is the source of power that both the theoretician and the engineer employ: manipulating a model in order to spawn predictions, noting particularly good and bad outcomes, and carrying out experiments to verify them. What sorts of models might be useful?

Consider a model of the program's user. For instance, there might be a rule that says "IF the user starts a sentence with the word LET, THEN assume he is a mathematician". There may be many rules already in the system that begin "IF the user is a mathematician..." and tell how to treat that group of people. Some of these rules, while important and useful, are at the superficial level of vocabulary , e.g., what the word "function" means to mathematicians as opposed to computer scientists, or physiologists. Other rules provide guidelines for what knowledge they can be assumed to know (and therefore would find insulting to be told), and what information they are unaware of (e.g., most mathematicians don't know what AI stands for.) Still other rules indicate what aspects of a problem-solving process will be of most interest to them (e.g., mathematicians are interested more in a final proof than in the pattern of blind alleys that led to it; psychologists are just the reverse). Finally, models of individual users can also be valuable, e.g. in noting and exploiting particular weaknesses in one's opponent at backgammon or chess.

Another useful type of model is a script of a typical usage of the program, scenarios of problems being solved in the task domain. These can provide additional sources of knowledge to interpret user inputs or to explain the various defaults the program is assuming. Eventually this can lead to a theory of problem solving in that domain.

7. Conclusions

We began by listing, as research objectives, a series of ever more ambitious types of meta-cognition, that is, capabilities of a program beyond straight "performance": explanation, learning, self-description, and finally self-modification (including self-redesign). We also divided meta-knowledge into three types: strategic, descriptive, and systemic.

Much of this paper has dealt with explanation, since it is the simplest sort of meta-cognition, the best-understood, and the only one in routine use in expert systems. Treating explanation separately in each section of this chapter may seem disconnected, but it is due to the multiple meanings of the English query "Why?" There is an interesting range of what that question can mean:

What is your immediate goal? (descriptive meta-knowledge. e.g., Shrdlu, Mycin)
What is an abstraction of your immediate goal? (descriptive meta-knowledge)
What is the justification for that? (descriptive meta-knowledge. e.g., Mycin)
Teleologically, how is it that that action contributes to the goal? (systemic)
Why is that a good choice of methods to accomplish that goal? (systemic)

Learning, self-description, and self-modification were returned to in the previous section of this chapter. "Learning", the incorporation of additional knowledge into expert systems, ranges from human data entry (learning by being told), to data gathering (learning by observing), to full-fledged theory formation (learning by discovery).

One important kind of learning is the compiling of descriptive meta-knowledge into strategic form, recasting it into a form in which it can be evaluated efficiently. Much of what we earlier called strategic meta-knowledge may be seen to be operationalized "caches" of descriptive or systemic meta-knowledge. For instance, R9 and R10 can be converted from systemic to strategic form by slight rewordings of their actions. In the other direction, R5 is the compiled strategic form of these two rules, the first of which is descriptive and the second is systemic:

```
R5a: The experts' spill rules are much much more reliable than the novices'.
R5b: In the long run, it pays for an expert system to use reliable rules.
```

Since knowledge breaks down into vocabulary, facts, and heuristics, so does learning:
(i) One can learn concepts in a fixed vocabulary. Unfortunately, this leads to performance which appears quite impressive but which is quite brittle and narrow. Early versions of Bacon [Langley 79] suffered from this problem, and one wondered what took Kepler, Balmer and the others so long to find their laws! Most AI research in learning and concept formation falls into this category.
(ii) One can learn useful new concepts and terms as well, periodically, thus increasing one's vocabulary, and then the range of learnable material is -- in theory -- unlimited. In his invited IJCAI6 address, in 1979, Herbert Simon discussed two programs (AM and Bacon.3) which are at this plateau.
(iii) Finally, one can learn new heuristics which will help efficiently search the new spaces engendered by the new concepts and terms. Eurisko [Lenat 82] carries out

this type of task.

Each of these three types of learning is qualitatively more costly in resources and program complexity than the preceding type. Eventually even the final type (learning new heuristics as well as new terms and new combinations of terms) may not suffice, as programs continue to increase in size and complexity:

Changing a huge program is currently a painful, often impossible, task. Static comments and documentation help. *Dynamic* documentation can be ever more useful: the program has available to it, explicitly, a model of its organization, overall design, detailed algorithms and knowledge representations, typical behaviors, etc. Both descriptive and systemic knowledge combine to provide this type of self-description capability for a program.

Static documentation requires manual updating whenever the program changes, but dynamic self-description ought to be computed by the program itself, and should change automatically with the program's executable code. A simple example of this is Interlisp's MasterScope [Teitelman and Masinter 81] facility: one can ask "Which functions call FIB7", e.g., and the answer will be correct even if you've just defined a new function which calls FIB7. As another example, the expert system we've grown in this chapter keeps track of the average running times and percentage of success of its rules.

Once self-description is a reality, the next logical step is self-modification. Small self-modifying automatic programming systems have existed for a decade, some large programs which modify themselves in very small ways also exist, and the first large fully self-describing and self-modifying programs are being built.

The capabilities of our machines, and the scope of our vision, has finally exceeded human cognitive capabilities in this dimension; it may now be cognitively economical to spend the energy building metacognition (especially self-description and self-modification) capabilities into large expert systems.

Acknowledgements

This paper began as a series of discussions at the 1980 Workshop on Expert Systems, in San Diego, organized by Hayes-Roth, Waterman, and Lenat. The output from that workshop is a book, to appear in mid-1982, of which this is one chapter. Davis was the note-taker at these discussions. Lenat reorganized the notes, added some additional material, and turned them into prose.

References

A. Barr and E. A. Feigenbaum, eds., *Handbook of AI,* William Kaufman, Palo Alto, 1980.

R. Balzer, L. Erman, P. London, and C. Williams, "HEARSAY-III: A domain-independent framework for expert systems", *Proc. 1st Annual National Conference on AI,* 1980, pp. 108-110.

D. R. Barstow, "An experiment in knowledge-based automatic programming", *AIJ* **12**, August, 1979, pp. 7-119.

B. G. Buchanan and T. Mitchell, *Model directed learning of production rules*, CS-77-597, Stanford University, March, 1977.

R. Burstall and J. Darlington , *A transformation system for developing recursive programs*, U. Edinburgh AI Research Report, March, 1976.

W. J. Clancey, E. H. Shortliffe, and B. G. Buchanan, "Intelligent computer-aided instruction for medical diagnosis", *Proc. 3rd Symposium on Computer Applications in Medical Care,* 1979, pp. 175-183.

R. Davis, "Interactive Transfer of Expertise: acquisition of new inference rules," *AIJ*, **12**, August, 1979, pp. 121-157.

R. Davis, "Meta-rules: Reasoning about control", *AIJ*, **15**, 1980.

R. Davis and D. B. Lenat, *Knowledge Based Systems in AI*, New York: McGraw Hill, 1981.

J. Doyle, *A Model for deliberation, action, and introspection,* Ph.D. thesis, MIT AI Lab, May, 1980.

R. O. Duda, *et al., A Computer-based Consultant for Minteral Exploration",* Final Report, SRI Project 6415, AI Center, SRI International, Menlo Park, September, 1979.

R. O. Duda, J. G. Gaschnig, and P. E. Hart, "Model design in the prospector consultant system for mineral explorations", in (D. Michie, ed.) *Expert Systems in*

the Microelectronic Age, Edinburgh: Edinburgh University Press, 1979, pp. 153-167.

E. A. Feigenbaum, "The Art of Artificial Intelligence", *Proc. 5th IJCAI*, August, 1977, MIT, pp. 1014-1029.

M. R. Genesereth, "Metaphors and Models", in *Proc. 1st AAAI Conf.*, Stanford, 1980, p. 208-211.

C. Green, R. Waldinger, D. Barstow, R. Eischlager, D. Lenat, B. McCune, D. Shaw, and L. Steinberg, *Progress Report on Program-Understanding Systems*, Memo AIM-240, CS Report 74-444, AI Lab, Stanford University, August, 1974.

R. Greiner and D. B. Lenat, "RLL: A representation language language", *Proc. 1st National Conf. on AI*, Stanford, 1980, pp. 165-169.

F. HayesRoth, P. Klahr, J. Burge, and J. Mostow, *Machine methods for acquiring, learning, and applying knowledge*, Report P6241, Rand Corporation, 1978.

Hempel, C. G., *Fundamentals of Concept Formation in Empirical Science*, U. Chicago Press, 1952.

C. K. Johnson and S. R. Jordan, "Inland oil and hazardous chemical spills: a systems knowledge-engineering problem", in Hayes-Roth, Waterman, and Lenat (eds), *Building Expert Systems*, forthcoming, 1982.

E. Kant, "A knowledge-based approach to using efficiency estimation in program synthesis", *Proc. 6th IJCAI*, Tokyo, 1979, pp. 457-462.

P. Langley, "Rediscovering physics with Bacon.3", *Proc. 6th IJCAI*, Tokyo, 1979, pp. 505-507.

D. B. Lenat, "Beings: Knowledge as interacting experts," *Proc. 4th IJCAI*, Tbilisi, 1975.

D. B. Lenat, "AM: Automated discovery in mathematics", *Proc. 5th IJCAI*, MIT, August, 1977.

D. B. Lenat, F. Hayes-Roth, and P. Klahr, *Cognitive Economy*, HPP Report HPP-79-15, Stanford University Computer Science Department, June, 1979.

D. B. Lenat, "The Nature of Heuristics", to appear in *AI Journal*, Summer, 1982.

J. R. Low, *Automatic Coding: Choice of Data Structures*, ISR 16, Birkheuser Verlag, 1976.

W. A. Martin and R. J. Fateman, "The MACSYMA System", *Proc. 2nd Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, 1971, pp. 59-75.

J. McCarthy, "Programs with common sense", reprinted in M. Minsky, ed., *Semantic*

Information Processing, Cambridge: MIT Press, 1968, pp. 403-410.

D. McDermott and J. Doyle, "Non-Monotonic Logic I", *AIJ,* **13**, 1-2, April, 1980.

R. Michalski, "A system of programs for computer-aided induction: a summary", *Proc. 5th IJCAI,* MIT, 1977, pp. 319-320.

G. Polya, *Mathematics and Plausible Reasoning,* Princeton U. Press, 1954.

C. Rich and H. Schrobe, *Intial Report on a LISP Programmer's Apprentice,* MIT AI Lab TR-354, 1976.

E. Shortliffe, *Computer Based Medical Consultations: MYCIN.* New York: Elsevier, 1976.

R. Stallman and G. J. Sussman, "Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis", *AIJ* 9, 1977, 135-196.

M. Stefik, J. Aikins, R. Balzer, J. Benoit, L .Birnbaum, F. Hayes-Roth, and E. Sacerdoti, "The architecture of expert systems", appear in Hayes-Roth, Waterman, and Lenat, eds., *Building Expert Systems,* forthcoming, 1982.

W. R. Swartout, *A Digitalis Adviser with Explanations,* MIT/LCS/TR-176, MIT, February, 1977.

W. Teitelman and L. Masinter, "The INTERLISP programming environment", *Computer,* **14**, 4, April, 1981, pp. 25-33.

S. A. Vere, "Induction of reliable productions in the presence of background information", *Proc. 5th IJCAI,* MIT, 1979, pp. 349-355.

D. Waterman and F. Hayes-Roth, eds., *Pattern Directed Inference Systems,* New York: Academic Press, 1978.

P. Winston, *Learning Structural Descriptions from Examples,* Ph.D. thesis, Dept. of Electrical Engineering, TR-76, MIT, September, 1970.