Designing a Rule System That Searches for Scientific Discoveries

Douglas B. Lenat
and
Gregory Harris

April 1977

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

ABSTRACT

Some scientific inference tasks (including mass spectrum identification [Dendral], medical diagnosis [Mycin], and math theory development [AM]) have been successfully modelled as rule-directed search processes. These rule systems are designed quite differently from "pure production systems". By concentrating upon the design of one program (AM), we shall show how 13 kinds of design deviations arise from *(i)* the level of sophistication of the task that the system is designed to perform, *(ii)* the inherent nature of the task, and *(iii)* the designer's *view* of the task. The limitations of AM suggest even more radical departures from traditional rule system architecture. All these modifications are then collected into a new, complicated set of constraints on the form of the data structures, the rules, the interpreter, and the distribution of knowledge between rules and data structures. These new policies sacrifice uniformity in the interests of clarity, efficiency and power derivable from a thorough characterization of the task. Rule systems whose architectures conform to the new design principles will be more awkward for many tasks than would "pure" systems. Nevertheless, the new architecture should be significantly more powerful and natural for building rule systems that do scientific discovery tasks.

1. The Basic Argument

Although rule-based computation was originally used for formal and systems purposes [Post,Markov,Floyd], researchers in Artificial Intelligence (AI) found that the same methodology was also useful for modelling a wide variety of sophisticated tasks. Many of these early AI rule-based programs -- called "production systems" -- served as information processing models of humans performing cognitive tasks in several domains (digit recall [19], algebra word problem solving [1], poker playing [23], etc. [16,18]).

There were many design constraints present in the classical formal rule based systems. Many of these details were preserved in the AI production rule based programs (e.g., forcing all state information into a single string of tokens). But there were many changes. The whole notion of "what a rule system really is" changed from an effective problem statement to a tendency to solve problems in a particular way. One typical corollary of this change of view was that instead of *no* external inputs whatsoever, there was now a *presumption* of some "environment" which supplied new entries into the token sequence. In the next section (see Figure 1) is an articulation of these *neo*-classical (i.e., AI circa 1973; see [7]) principles for designing "pure" production systems.

Due to the early successes, psychological applicability, and aesthetic simplicity afforded by production systems, AI researchers began to write rule systems (RSs) to perform informal inductive inference tasks (mass spectrum identification [4], medical diagnosis [23] and consultation dialogue [6], speech understanding [14], non-resolution theorem proving [0], math research [13], and many more).

Yet it seems that most of the large, successful RSs have violated many of the "pure production system" guidelines. The purpose of this paper is to show that such "exceptions" were inevitable, because any system satisfying the neo-classical design constraints, though universal in principle, is too impoverished to represent complex tasks for what they are.

The essence of the neo-classical architecture is to opt for simplicity in all things, since there is very little one can say about RSs in general. As more becomes known about the task of the RS, it turns out that some of that new knowledge takes the form of specific constraints on the design of the RS itself (as distinct from what specific knowledge we choose to represent within that design). Sometimes a new constraint directly contradicts the early, domain-independent one; sometimes it is merely a softening or augmentation of the old constraint.

After examining the "pure" architecture, we shall examine in detail the design of one particular rule system which discovers and studies mathematical concepts. Deviations from the pure architecture will be both frequent and extreme.

Subsequent sections will analyze these differences. It will be shown that each one is plausible -- usually for reasons which depend strongly on the "scientific discovery"

domain of the RS. Some of the limitations of this RS will be treated, and their elimination will be seen to require abandoning still more of the original design constraints.

When these modifications are collected, in the final section, we shall have quite a different set of principles for building RSs. Not only will naivete have been lost: so will generality (the breadth of kinds of knowledge representable, the totality of tractable tasks). Rule systems conforming to the new design will be awkward for many tasks (just as a sledge hammer is awkward for cracking eggs). However, they should be significantly more powerful and natural for scientific inference tasks.

2. Early Design Constraints

By a *rule system* (RS) we shall mean any collection of condition-action *rules*, together with associated *data structures* (DS; also called *memories*) which the rules may inspect and alter. There must also be a policy for *interpretation*: detecting and firing relevant rules.

These definitions are deliberately left vague. Many details must be specified for any actual rule system (e.g., What may appear in the condition part of a rule?). This specification process is what we mean by *designing* a RS.

Figure 1 contains an articulation of the design of the early general-purpose AI production rule systems. Notice the common theme: the adequacy of simplicity in all dimensions.

FIGURE 1: Neo-classical Rule System Architecture

1. *Principle of Simple Memories. One or two uniform data structures define sufficient memories for a rule system to read from and write into. The format for entries in these structures is both uncomplicated and unchanging.*

2. *Principle of Simple DS Accesses. The primitive read and write operations are as simple and low-level as possible; typically they are simply a membership-test type of read, and an insert-new-element type of write. More complicated, algorithmic operations on the memories are not available to the rules.*

3. *Principle of Isolated DS Elements. Elements of the uniform DS cannot point to (parts of) other elements. This follows from the preceding principle: if we aren't allowed to chase pointers, there may as well not be any.*

4. *Principle of Continuous Attention. In addition to the one or two simple data structures, there may be an external environment which continuously inserts stimuli into the DS. The interleaving of stimuli and internally generated symbols is managed quite trivially: (a) The stimuli are simply inserted into*

the DS as new elements; (b) Each rule is so small and quick that no "interruption" mechanism is necessary. The interpreter may ignore any suddenly-added stimulus until the current rule finishes executing. The RS may be viewed as "continuously" attending to the environment.

5. *Principle of Opaque Rules. Rules need not have a format inspectable by other rules, but rather can be coded in whatever way is convenient for the programmer and the rule interpreter; i.e., the set of rules is <u>not</u> treated as one of the RSs data structures. E.g., the condition parts of rules may be barred from fully analyzing the set of productions [22], and the action parts of rules may not be allowed to delete existing rules [24].*

6. *Principle of Simple Rules. Rules consist of a left- and a right-hand side which are quite elementary: The left hand side (lhs, situation characterization, IF-part, condition) is typically a pattern-match composed with a primitive DS read access, and the right hand side (rhs, consequence, THEN-part, action) is also simply a primitive DS write access. There is no need for sophisticated bundles of DS accesses on either side of a rule. Thus several extra rules should be preferred to a single rule with several actions.*

7. *Principle of Encoding by Coupled Rules. A collection of interrelated rules is used to accomplish each subtask; i.e., wherever a subroutine would be used in a procedural programming language. For example, programming an iteration may require many rules "coupled" by writing and reading special (i.e., otherwise meaningless) loop control notes in the data structure.*

8. *Principle of Knowledge as Rules. All knowledge of substance should be, can be, and is represented as rules. This includes all non-trivial domain-dependent information. The role of the DS is just to hold simple descriptive information, intermediate control state messages, recent stimuli from the environment, etc.*

9. *Principle of Simple Interpretation. The topmost control flow in the RS is via a simple rule interpreter. After a rule fires, it is essential that any rule in the system may potentially be the next one to fire (i.e., it is forbidden to locate a set of relevant rules and fire them off in sequence). When the rhs of a rule is executed, it can (and frequently will) drastically alter the situation that determined which rules were relevant.*

10. *Principle of Closure. The representations allowed by (1-9) are sufficient and appropriate for organizing all the kinds of knowledge needed for tasks for which a given RS is designed.*

This design was plausible *a priori*, and worked quite well for its initial applications (the simulation of simple human cognitive processes [16,19,24]). But is this design proper for *any* RS, regardless of its intended task? In particular, what about scientific inference tasks? Over the years, several rule-based inference systems for scientific tasks have been constructed. With each new success have come some deviations from

the above principles [7]. Were these mere aberrations, or is there some valid reason for such changes in design?

We claim the latter. The task domain -- scientific discovery -- dictates a new and quite different architecture for RSs. To study this phenomenon, we shall describe, in the next section, one particular RS which defines new mathematical concepts, studies them, and conjectures relationships between them. Subsequent sections will explore the deviations of its design from the neo-classical constraints in Figure 1.

3. 'AM': A Rule System For Math Theory Formation

A recent thesis [13] describes a program, called "AM", which gradually expands a base of mathematical knowledge. The representation of math facts is somewhat related to Actors [10] and Beings [12] in the partitioning of such domain knowledge into effective, structured modules. Departing from the traditional control structures usually associated with Actors, Beings, and Frames [15], AM concentrates on one "interesting" mini-research question after another. These "jobs" are proposed by -- and rated by -- a collection of approximately 250 situation-action rules. Discovery in mathematics is modelled in AM as a rule-guided exploration process. This view is explained below in Section 3.1 (See also [21].) The representation of knowledge is sketched next, followed by a much more detailed description of the rule-based control structure of AM. Finally, in Section 3.5, the experimental results of the project are summarized.

3.1. Discovery in Mathematics as Heuristic Rule-Guided Search

The task which AM performs is the discovery of new mathematics concepts and relationships between them. The simple paradigm it follows for this task is to maintain a graph of partially-developed concepts , and to obey a large collection of "heuristics" (rules which frequently lead to discoveries) which guide it to define and study the most plausible thing next.

For example, at one point AM had some notions of sets, set-operations, numbers, and simple arithmetic. One heuristic rule it knew said "*If f is an interesting relation, Then look at its inverse*". This rule fired after AM had studied "multiplication" for a while. The rhs of the rule then directed AM to define and study the relation "divisors-of" (e.g., divisors-of(12) = {1,2,3,4,6,12}). Another heuristic rule which later fired said "*If f is a relation from A into B, then it's worth examining those members of A which map into extremal members of B*". In this case, f was matched to "divisors-of", A was "numbers", B was "sets of numbers", and an extremal member of B might be, e.g., a very *small* set of numbers. Thus this heuristic rule caused AM to define the set of numbers with no divisors, the set of numbers with only 1 divisor, with only 2 divisors, etc. One of these sets (the last one mentioned) turned out subsequently to be quite important; these numbers are of course the primes. The above heuristic also directed AM to study numbers with very *many* divisors; such highly-composite numbers were also found to be interesting.

This same paradigm enabled AM to discover concepts which were much more primitive (e.g., cardinality) and much more sophisticated (e.g., the fundamental theorem of arithmetic) than prime numbers. We shall now describe the AM program in more detail.

3.2. Representation of Mathematical Knowledge

What exactly does it mean for AM to "have the notion of" a concept? It means that AM possesses a frame-like data structure for that concept. For instance, here is how one concept looked after AM had defined and explored it:

FIGURE 2: A Typical Concept

NAME: Prime Numbers
DEFINITIONS:
ORIGIN: Number-of-divisors-of(x) = 2
PREDICATE-CALCULUS: Prime(x) ≡ (∀z)(z|x → z=1 XOR z=x)
ITERATIVE: (for x>1): For i from 2 to x-1, ¬(i|x)
EXAMPLES: 2, 3, 5, 7, 11, 13, 17
BOUNDARY: 2, 3
BOUNDARY-FAILURES: 0, 1
FAILURES: 12
GENERALIZATIONS: Numbers, Numbers with an even number of divisors,
Numbers with a prime number of divisors
SPECIALIZATIONS: Odd Primes, Prime Pairs, Prime Uniquely-addables
CONJECS: Unique factorization, Goldbach's conjecture, Extrema of Divisors-of
ANALOGIES:
Maximally-divisible numbers are converse extremes of Divisors-of
INTEREST: Conjectures tying Primes to Times, to Divisors-of, to closely related ops
WORTH: 800

3.3. Top-level Control: An Agenda of Promising Questions

AM was initially given a collection of 115 core concepts, with only a few facets (i.e., slots) filled in for each. AM repeatedly chooses some facet of some concept, and tries to fill in some entries for that particular slot. To decide which such job to work on next, AM maintains an *agenda* of jobs, a global queue ordered by priority [2]. A typical job is "*Fill-in examples of Primes*". The agenda may contain hundreds of entries such as this one. AM repeatedly selects the top job from the agenda and tries to carry it out. This is the whole control structure! Of course, we must still explain how AM creates plausible new jobs to place on the agenda, how AM decides which job will be the best one to execute next, and how it carries out a job.

If the job were "*Fill in new Algorithms for Set-union*", then *satisfying* it would mean actually synthesizing some new procedures, some new LISP code capable of forming the union of any two sets. A heuristic rule is *relevant* to a job if and only if executing

that rule brings AM closer to satisfying that job. Potential relevance is determined *a priori* by where the rule is stored. A rule tacked onto the Domain/range facet of the Compose concept would be presumed potentially relevant to the job "*Fill in the Domain of Insert-o-Delete*". The lhs of each potentially relevant rule is evaluated to determine whether the rule is truly relevant.

Once a job is chosen from the agenda, AM gathers together all the potentially relevant heuristic rules -- the ones which might accomplish that job. They are executed, and then AM picks a new job. While a rule is executing, three kinds of actions or effects can occur:

(i) Facets of some concepts can get filled in (e.g., examples of primes may actually be found and tacked onto the "Examples" facet of the "Primes" concept). A typical heuristic rule which might have this effect is:

> *If examples of X are desired, where X is a kind of Y (for some more general concept Y),*
> *Then check the examples of Y; some of them may be examples of X as well.*

For the job of filling in examples of Primes, this rule would have AM notice that Primes is a kind of Number, and therefore look over all the known examples of Number. Some of those would be primes, and would be transferred to the Examples facet of Primes.

(ii) New concepts may be created (e.g., the concept "primes which are uniquely representable as the sum of two other primes" may be somehow be deemed worth studying). A typical heuristic rule which might result in this new concept is:

> *If some (but not most) examples of X are also examples of Y (for some concept Y),*
> *Then create a new concept defined as the intersection of those 2 concepts (X and Y).*

Suppose AM has already isolated the concept of being representable as the sum of two primes in only one way (AM actually calls such numbers "Uniquely-prime-addable numbers"). When AM notices that some primes are in this set, the above rule will create a brand new concept, defined as the set of numbers which are both prime and uniquely prime addable.

(iii) New jobs may be added to the agenda (e.g., the current activity may suggest that the following job is worth considering: "Generalize the concept of prime numbers"). A typical heuristic rule which might have this effect is:

> *If very few examples of X are found,*
> *Then add the following job to the agenda: "Generalize the concept X".*

The concept of an agenda is certainly not new: schedulers have been around for a long time. But one important feature of AM's agenda scheme *is* a new idea: attaching -- and

using -- a list of quasi-symbolic reasons to each job which explain why the job is worth considering, why it's plausible. It is the responsibility of the heuristic rules to include reasons for any jobs they propose. For example, let's reconsider the heuristic rule mentioned in *(iii)* above. It really looks more like the following:

> *If very few examples of X are found,*
> *Then add the following job to the agenda: "Generalize the concept X", for the following reason: "X's are quite rare; a slightly less restrictive concept might be more interesting".*

If the same job is proposed by several rules, then several different reasons for it may be present. In addition, one ephemeral reason also exists: "Focus of attention" [9]. Any jobs which are related to the one last executed get "Focus of attention" as a bonus reason. AM uses all these reasons to decide how to rank the jobs on the agenda. Each reason is given a rating (by the heuristic which proposed it), and the ratings are combined into an overall priority rating for each job on the agenda. The jobs are ordered by these ratings, so it is trivial to select the job with the highest rating. Note that if a job already on the agenda is re-proposed for a new reason, then its priority will increase. If the job is re-proposed for an already-present reason, however, the overall rating of the job will *not* increase. This turned out to be an important enough phenomenon that it was presented in [13] as a necessary design constraint.

AM uses each job's list of reasons in other ways. Once a job has been selected, the quality of the reasons is used to decide how much time and space the job will be permitted to absorb, before AM quits and moves on to a new job. Another use is to explain to the human observer precisely why the chosen top job is a plausible thing for AM to concentrate upon.

3.4. Low-level Control: A Lattice of Heuristic Rules

The hundreds of concepts AM possesses are interrelated in many ways. One main organization is that provided by their Generalization and Specialization facets. The concepts may be viewed as nodes on a large lattice whose edges are labelled Genl/Spec. The importance of this organization stems from various *heritability* properties. For example, Spec is transitive, so the specializations of Numbers include not only Primes but all *its* specializations as well.

Let us describe a second, very important heritability property. Each of the 250 heuristic rules is attached to the most general (or abstract) concept for which it is deemed appropriate. The relevance of heuristic rules is assumed to be inherited by all its specializations. For example, a heuristic method which is capable of inverting any function will be attached to the concept "Function"; but it is certainly also capable of inverting any permutation. If there are no known methods specific to the latter job, then AM will follow the Genl links upward from Permutation to Bijection to Function..., seeking methods for inversion. Of course the more general concepts' methods tend to be weaker than those of the specific concepts.

In other words, the Genl/Spec graph of concepts induces a graph structure upon the set of heuristic rules. This permits potentially relevant rules to be located efficiently. Here is one more example of how this heritability works in practice: Immediately after the job "Fill in examples of Set-equality" is chosen, AM asks each generalization of Set-equality for help. Thus it asks for ways to fill in examples of any Predicate, any Activity, any Concept, and finally for ways to fill in examples of Anything. One such heuristic rule known to the Activity concept says: "*If examples of the domain of the activity f are already known, Then actually execute f on some random members of its domain.*" Thus when AM applies this heuristic rule to fill in examples of Set-equality, its Domain facet is inspected, and AM notes that Set-equality takes a pair of sets as its arguments. Then AM accesses the Examples facet of the concept Set, where it finds a large list of sets. The lhs is thus satisfied, so the rule is fired. Obeying the heuristic rule, AM repeatedly picks a pair of the known sets at random, and sees if they satisfy Set-equality (by actually running the LISP function stored in the Algorithms facet of Set-equality). While this will typically return False, it will occasionally locate -- by random chance -- a pair of equal sets.

Other heuristics, tacked onto other generalizations of Set-equality, provide additional methods for executing the job "Fill in examples of Set-equality." A heuristic stored on the concept Any-concept says to symbolically instantiate the definition. After spending much time manipulating the recursive definition of Set-equality, a few trivial examples (like {}={}) are produced. Notice that (as expected) the more general the concept is, the weaker (more time-consuming, less chance for success) its heuristics tend to be. For this reason, AM consults each concept's rules in order of increasing generalization.

3.5. Behavior of this Rule System

As the preceding four sections indicate, the dynamic behavior of AM was as follows: a job is chosen from the agenda, potentially relevant rules are located by their position in the Genl/Spec lattice, their lhs's (left-hand sides) are evaluated to find those which actually trigger, they are then executed (in order of decreasing specificity) until they are all executed (or until some *local*, self-imposed limit on time or space is exceeded), and the cycle repeats. AM has a modest facility that prints out a description of these activities as they occur. Here is a tiny excerpt of this self-trace monologue.

** Job 65: ** Fill in Examples of the concept "Divisors-of".

3 Reasons: (1) No known examples of Divisors-of so far.
(2) TIMES, which is related to Divisors-of, is now very interesting.
(3) Focus of attention: AM recently defined Divisors-of..

26 examples found, in 9.2 seconds. e.g., Divisors-of(6)={1 2 3 6}.

** Job 66: ** Consider numbers having small sets of Divisors-of.

2 Reasons: (1) Worthwhile to look for extreme cases.
(2) Focus of attention: AM recently worked on Divisors-of.

Filling in examples of numbers with 0 divisors.
0 examples found, in 4.0 seconds.
Conjecture: no numbers have precisely 0 divisors.

Filling in examples of numbers with 1 divisors.
1 examples found, in 4.0 seconds. e.g., Divisors-of(1) = {1}.
Conjecture: 1 is the only number with precisely 1 divisor.

Filling in examples of numbers with 2 divisors.
24 examples found, in 4.0 seconds. e.g., Divisors-of(13) = {1 13}.
No obvious conjecture. May merit more study.
Creating a new concept: "Numbers-with-2-divisors".

Filling in examples of numbers with 3 divisors.
11 examples found, in 4.0 seconds. e.g., Divisors-of(49) = {1 7 49}.
All numbers with 3 divisors are also Squares. Definitely merits more study.
Creating a new concept: "Numbers-with-3-divisors".

** Job 67: ** Consider the square-roots of Numbers-with-3-divisors.

2 Reasons: (1) Numbers-with-3-divisors are unexpectedly also Perfect Squares.
(2) Focus of attention: AM recently worked on Nos-with-3-divisors.

All square-roots of Numbers-with-3-divisors seem to be Numbers-with-2-divisors.
e.g., Divisors-of(Square-root(169)) = Divisors-of(13) = {1 13}.
Even the converse of this seems empirically to be true.
i.e., the square of each No-with-2-divisors seems to be a No-with-3-divisors.
The chance of coincidence is below acceptable limits.
Boosting the interestingness rating of each of the concepts involved.

** Job 68: ** Consider the squares of Numbers-with-3-divisors.

3 Reasons: (1) Squares of Numbers-with-2-divisors were interesting.
(2) Square-roots of Numbers-with-3-divisors were interesting.
(3) Focus of attention: AM recently worked on Nos-with-3-divisors.

Now that we've seen how AM works, and we've been exposed to a bit of "local" results, let's take a moment to discuss the totality of the mathematics which AM carried out. AM began its investigations with scanty knowledge of a hundred elementary concepts of finite set theory. Most of the obvious set-theoretic concepts and

relationships were quickly found (e.g., de Morgan's laws; singletons), but no sophisticated set theory was ever done (e.g., diagonalization). Rather, AM discovered natural numbers and went off exploring elementary number theory. Arithmetic operations were soon found (as analogs to set-theoretic operations), and AM made surprising progress in divisibility theory. Prime pairs, Diophantine equations, the unique factorization of numbers into primes, Goldbach's conjecture -- these were some of the nice discoveries by AM. Many concepts which we know to be crucial were never uncovered, however: remainder[1], gcd, greater-than, infinity, proof, etc.

All the discoveries mentioned were made in a run lasting one cpu hour (Interlisp+100k, SUMEX PDP-10 KI). Two hundred jobs in toto were selected from the agenda and executed. On the average, a job was granted 30 cpu seconds, but actually used only 18 seconds. For a typical job, about 35 rules were located as potentially relevant, and about a dozen actually fired. AM began with 115 concepts and ended up with three times that many. Of the synthesized concepts, half were technically termed "losers" (both by the author and by AM), and half the remaining ones were of only marginal interest.

Although AM fared well according to several different measures of performance (see Section 7.1 in [13]), of greatest significance are its *limitations*. This subsection will merely report them, and the next section will analyze whether they were caused by radical departures from the neo-classical production-system architecture, or from departing not far enough from that early design.

As AM ran longer and longer, the concepts it defined were further and further from the primitives it began with. Thus "prime-pairs" were defined using "primes" and "addition", the former of which was defined from "divisors-of", which in turn came from "multiplication", which arose from "addition", which was defined as a restriction of "union", which (finally!) was a primitive concept (with heuristics) that we had supplied to AM initially. When AM subsequently needed help with prime pairs, it was forced to rely on rules of thumb supplied originally about *union*ing. Although the heritability property of heuristics did ensure that those rules were still valid, the trouble was that they were too general, too weak to deal effectively with the specialized notions of primes and arithmetic. For instance, one general rule indicated that AUB would be interesting if it possessed properties absent both from A and from B. This translated into the prime-pair case as "*If p+q=r, and p,q,r are primes, Then r is interesting if it has properties not possessed by p or by q.*" The search for categories of such interesting primes *r* was of course barren. It showed a fundamental lack of understanding about numbers, addition, odd/even-ness, and primes.

As the derived concepts moved further away from finite set theory, the efficacy of the initial heuristics decreased. AM began to "thrash", appearing to lose most of its heuristic guidance. It worked on concepts like "prime triples", which is not a rational thing to investigate. The key deficiency was the lack of adequate *meta*-rules[6]: heuristics which cause the creation and modification of new heuristics.

---

[1] This concept, and many of the other "omissions", could have been discovered by the existing heuristic rules in AM. The paths which would have resulted in their definition were simply never rated high enough to explore.

Aside from the preceding major limitation, most of the other problems pertain to missing knowledge. Many concepts one might consider basic to discovery in math are absent from AM; analogies were under-utilized; physical intuition was absent; the interface to the user was far from ideal; etc.

4. Reexamining the Design

Let us now consider the major components of a RS's design and how AM treated them: the DS, the rules, the distribution of knowledge between DS and rules, and the rule interpretation policy. For each component, AM's architecture failed to adhere strictly to the pure RS guidelines. Were these departures worth the loss of simplicity? Were the deviations due to the task domain (scientific discovery), to the task view (heuristically guided growth of structured theories), or to other sources? These are the kinds of questions we shall address in each of the following subsections.

4.1. Data Structures

We recognize that a single uniform DS (e.g., an infinite STM [19]) is universal in the Turing sense of being *formally* adequate: One can encode any representation in a linear, homogeneous DS. The completeness of such a DS design not withstanding, we believe that encouraging several distinct, special-purpose DSs will enhance the performance of a discovery system. That is, we are willing to sacrifice aesthetic purity of DSs for clarity, efficiency, and power. In this section we will explore this tradeoff.

The data structures used in AM are unlike the uniform memories suggested by the first design constraint (see Figure 1). One DS -- the agenda -- holds an ordered list of plausible questions for the system to concentrate on, a list of jobs to work on. Another DS is the graph of concepts AM knows about. Each concept itself consists in much structured information (see Figure 2). The reasons AM has for each job have information associated with them. Still other information is present as values of certain functions and global variables: the cpu clock, the total number of concepts, the last thing typed out to the user, the last few concepts worked on, etc. All these types of information are accessed by the lhs's (left hand sides) of heuristic rules, and affected by rhs's (some "deliberately" in the text of the rule, some "incidentally" through a chain of if-added methods).

Why is there this multitude of diverse DSs? Each type of knowledge (jobs, math knowledge, system status) needs to be treated quite differently. Since the primitive operations will vary with the type of information, so should the DS. For jobs, the primitive kinds of accesses will be: picking the highest-rated job, deleting the lowest-rated one, reordering some jobs, merging new ones. A natural choice to make these operations efficient is to keep the system's goals in a queue ordered by their rating or partially-ordered by those ratings that are commensurable. For resource information,

the usual request is for some *statistic* of some class of primary data. To maintain a table of such summary facts (like how much the CPU clock has run so far, or how many concepts there are) is to introduce an unnecessary DS and incur exorbitant costs to maintain many *short-lived* entries that will, most probably, never be used. It is far more reasonable to run a summarizing procedure to develop just that ephemeral, up-to-date information that you need. For math concepts, we have a much less volatile situation. We view them as an ever-growing body of highly-interrelated facts. Knowledge in this form is stable and rarely deleted. When new knowledge is added, a great many "routine" inferences must be drawn. In a uniform, linear memory, each would have to be drawn explicitly; in a structured one (as the Genl/Spec graph structure provides) they may be accomplished through the tacit (analogical) characteristics of the representation, simply by deciding *where* to place the information.

Each kind of knowledge dictates a set of appropriate kinds of primitive operations to be performed on it, which in turn suggest natural data structures in which to realize it. The generality of this perspective on rule-based systems is made more plausible by examining other RSs which deal with many types of knowledge (e.g., [5]). If this is so, if the design proceeds from "knowledge to be represented" to "a data structure to hold it", then fixing *a priori* the capabilities of the DS access primitives available to rules is suspect.

Therefore, we advocate the opposite: the RS designer is encouraged to name every combination of "machine" operations that together comprise a single conceptual access of data by rules. In AM, it is quite reasonable to expect that a request like "find all generalizations of a given concept" would be such a primitive (i.e., could be referred to by name). Even though it might cause the "machine" (in this case, LISP) to run around the Genl/Spec graph, a single rule can treat this as merely an "access" operation. The use of complex tests and actions is not new; we simply claim that it is *always* preferable to package knowledge (for which a reasonably fast algorithm is available) as a single action (though it may have side-effects in the space of concepts) or a single test (so long as its sole side-effect -- modulo caches -- is to signal). Primitive tests and actions should be maximally algorithmic, not minimally computational.

The neo-classical view of designing a production rule system was that of defining a machine. Our present view is that RSs do not *compute* so much as they *guide attention*. In adopting this view (thereby separating the controller from the effector), we recognize that we are giving up an attractive feature of pure rule systems: a homogeneous basis for definition. For example, the rule system designer must now spell out in detail the definitions of the DS accessing functions; but the designer of a neo-classical RS is simply able to take as *givens* the matching and inserting operations (as specified in neo-classical principle #6, Figure 1), and he builds each more complicated one out of these primitives[2]. In giving up the old view of the RS as an abstract computing machine, the RS designer must use another homogeneous substrate

---

[2] Either by stringing out a sequence of primitives on one side of a rule, or by handcrafting a tightly coupled bundle of rules (so firing such a rule would simulate traversing one link of the kind that abound in AM's DSs).

(e.g., LISP) in terms of which to define his DSs and especially the procedures that process them. In exchange, he obtains a clear distinction between two kinds of knowledge contained in the neo-classical rule: plausible proposals for what to do next, and how to accomplish what might be proposed.

We have seen that admitting complicated and varied DSs leads to stylized sets of DS accesses. The DSs and their sets of read/write primitives must in turn be explicitly defined (coded) by the designer. This seems like a high price to pay. Is there any bright side to this? Yes, one rather interesting possibility is opened up. Not only the RS designer, but the RS *itself* may define DSs and DS access functions. In AM, this might take the form of dynamically defining new kinds of facets (slots). E.g., after "injective Function" is defined, and after some properties of it have been discovered, it would be appropriate to introduce a new facet called "inverse" for each (concept representing an) injective function. In AM, the actual definitions of the facets of every concept are complex enough (shared structure), inter-related enough (shared meaning), and interesting enough (consistent heuristic worth) that a special concept was included for each one (e.g., a concept called "Examples") which contained a definition, description,... of the facet. Thus the same techniques for manipulating and discovering math concepts may be applied to DS design concepts. Not only do math theories emerge, so can new DS access functions (new slots; e.g., "Small Boundary Examples", "Factorization", or "Inverse").

It should be noted that in opting for non-uniform DSs, we have not in general sacrificed efficiency. One has only to compare the time to access a node in a tree, versus in a linear list, to appreciate that efficiency may, in fact, be *increased* by non-uniformity.

Just how tangled up a DS should we tolerate? Should memory elements be permitted to refer to (to "know about") each other? We believe the answer to depend upon the *type* of data structure involved. For the homogeneous DS called for in the neo-classical design, much simplicity is preserved by forbidding this kind of interrelationship. But consider a DS like AM's graph of concepts. It is growing, analogically interrelated, and it contains descriptions of its elements. This richness (and sheer quantity) of information can be coded only inefficiently in a uniform, non-self-referential manner. For another example, consider AM's agenda of jobs. One reason for a job may simply be the existence of some other job. In such a case, it seems natural for part of one entry on the agenda (a reason part of one job) to point to another entry in the same DS (point to another specific job on the agenda). Thus, inter-element pointers *are* allowed, even though they blur a "pure" distinction between a DS and its entries.[3] Inter-element references play a necessary role in organizing large bodies of highly interrelated information into structured modules.

There is yet another motivation for special-purpose DSs when the task of the RS includes sensing an external environment. Using a uniform memory, external stimuli are dumped into the working memory and rub shoulders with all the other data. They

---

[3] In section 4.3 we will mention work that blurs this distinction even further.

must then be distinguished from the others. ("Must" because to freely intermingle what one sees or is told with what one thinks or remembers is to give way to endless confusion.) How much cleaner, less distracting, and safer it is for stimuli to arive in their own special place -- a place which might well be a special purpose store such as an intensity *array* (not even a list structure at all), or a low-level speech-segment *queue*. A linear memory (e.g., an infinite STM) is of course adequate; one could tag each incoming environmental stimulus with a special flag. But the design philosophy we are proposing is aimed at maximizing clarity and efficiency, not uniformity or universality.

We know that this view of DSs means making a specialized design effort for each class of knowledge incorporated into the RS. But that is desirable, as it buys us three things: *(i)* system performance is increased, *(ii)* some forms of automatic learning are facilitated, *(iii)* knowledge is easier to encode.

4.2. Rules

In the "pure" view of RSs, the rule store is not a full-fledged DS of the RS. For example, in Waterman's [24] poker player, rules may not be deleted. Rychener [22] states that the only way his RS may inspect rules is by examining the effect of those rules which have recently fired. Although AM had no explicit taboo against inspecting rules, such analyses were in practice never possible, since the rules were *ad hoc* blocks of LISP code. This eventually turned out to be the main limitation of the design of AM. The ultimate impediment to further discovery was the lack of rules which could reason about, modify, delete, and synthesize other rules. AM direly needed to synthesize specialized forms of the given general heuristic rules (as new concepts arose; see the end of 3.5.)

We want our heuristic rules to be added, kept track of, reasoned about, modified, deleted, generalized, specialized, ... whenever there is a good reason to do so. Note that those situations may be very different from the ones in which such a rule might fire. E.g., upon discovering a new, interesting concept, AM should try to create some specially-tailored heuristic rules for it. They wouldn't actually *fire* until much later, when their lhs's were triggered. After having constructed such rules, AM might subject them to criticism and improvement as it explores the new concept.

In sum, we have found that the discovery of heuristic rules for using new math concepts is a necessary part of the growth of math knowledge. Hence, following the argument in 4.1, the rules themselves should be DSs, and each rule might be described by a concept with effective (executable) and non-effective (purely descriptive) facets. This lesson was made all the more painful because it was not new [5]. Apparently the need for reasoning about rules is common to many tasks.

The current re-coding of AM does in fact have each rule represented as a concept. What kinds of non-effective "facets" do they have? Recall that one of the features of the original AM (as described in Section 3.3) was that with each rule were associated some symbolic *reasons* which it could provide whenever it proposed a new job for the

agenda. So one kind of facet which every rule can possess is "Reasons". What others are there? Some of them *describe* the rule (e.g., its average cost); some facets provide a road map to the space of rules (e.g., which rule schemata are mere specializations of the given one); some facets record its derivation (e.g., the rule was proposed as an analog to rule X because ...), its redundancy (some other rules need not be tried if this one is), etc.

There are some far-reaching consequences of the need to reason about rules just as if they were any other concepts known to AM. When one piece of knowledge relates to several rules, then one general concept, a rule *schema*, should exist to hold that common knowledge. Since each rule is a concept, there will be a natural urge to exploit the same Genl/Spec organization that proved so useful before. Heritability still holds; e.g., any reason which explains rule R is also somehow a partial explanation of each specialization of R.

Rule schemata have cause to exist simply because they generalize -- and hold much information which would otherwise have to be duplicated in -- several specific rules. They may tend to be "big" and less directly productive when executed, yet they are of value in capturing the essence of the discovery techniques.[4] We put "big" in quotes because sheer length (total number of lhs tests allowed, total number of rhs actions) is not directly what we're talking about here. A general rule schema will capture many regularities, will express an idea common to several more specific rules. It will contain dual forms of the same rule, sophisticated types of variable-binding (for the duration of the rule application), and searching may even be required to find the actions of such a general rule. We may even wish to consider every rule in the RS as a rule schema of some level of generality, and much processing may go on to find the particular instance(s) of it which should be applied in any particular situation.

Let us consider a rule schema called the "rule of enthusiasm". It subsumes several rules in the original AM system (pp. 247-8 of [13]), e.g., those that said:

> *If concept* G *is now very interesting, and* G *was created as a generalization of some earlier concept* C,
> *Give extra consideration to generalizing* G, *and to generalizing* C *in other ways.*

and:

> *If concept* S *proved to be a dead-end, and* S *was created as a specialization of some earlier concept* C,
> *Give less consideration to specializing* S, *and to specializing* C *in other ways in the future.*

---

[4] In AM, even the specific rules may be "big" in the sense that their very precise knowledge may involve much testing to trigger and, once triggered, may conclude some elaborate results.

The proposed rule schema is:

> *If concept X has very high/low interest and X can be derived from some concept C by means m,*
> *Give more/less consideration to finding (and elaborating) concepts derived from C, X (and their "neighbors") by means analogous to m.*

There are four variables to be matched and coordinated in the lhs of this rule: a concept X, the direction (high or low) of its extreme interest rating, a derivation procedure m and an associated source concept C. The action itself is to search for jobs of a certain type and give them a corresponding (high or low) rating change. Three types of matching are present: *(i)* ranging over a set of alternatives which are known at the time the rule is written (e.g., the "high/low" alternative); *(ii)* ranging over a set of alternatives which can be accessed easily at any moment the rule is run, like the set of concepts and connections between them now in existence (e.g., the variables X and C range over this kind of set); *(iii)* ranging over a set of alternatives which must be heuristically searched for as part of the rule execution (e.g., "analogous" and "neighbors" only make sense after a nontrivial amount of searching has been performed).

Since the "rule of enthusiasm" is very general, it will only be tried if no more specific rules (such as the two which were listed just above it) are relevant at the time. Ideally, the search to specify the action should create a new, specialized form of the rule of enthusiasm to catch this situation and handle it quickly, should it arise again. Note that versions of this schema that mention generalization or specialization are also schemata (without any specification search); they are simply less general schemata than the rule of enthusiasm itself. Whenever a new subject for discovery gets defined, the abstract, hard-to-execute rule schemata can be specialized (compiled, refined , etc.) into efficient heuristics for that subject.

Another use of a rule schema might be to *name* a collection of neo-classical rules that are coupled by together fulfilling a single function. Consider a collection of rules which is tightly coupled, say to perform an iteration. Much knowledge about the iteration loop as a whole may exist. Where is such descriptive information to be stored and sought? Either it must be duplicated for each of the coupled rules, or there must be a rule-like concept which "knows about" the iteration as one coherent unit. We conclude that even if some intertwined rules *are* kept separate, an extra rule (a schema) should exist which (at least implicitly) has a rhs which combines them (by containing knowledge common to all of them). Thus rule schemata do more than just unify general properties of rules: there must also be schemata of the kind that relate function to mechanism.

Another problem crops up if we consider what happens if one of the coupled rules is modified. Often, some corresponding change should be made in all its companions. For example, if a term is generalized (replacement of "prime" by "number" everywhere) then the same substitution had probably better be done in each rule with which this one is supposed to couple. What we are saying is that, for RSs which modify their own rules, it can be dangerous to split up a single conceptual process into a bunch of rules which interact in more or less fixed ways when run, without continuing to reason

about them as an integrity, *like any other algorithm* composed of parts. Here again, we find pressure to treat RSs as algorithms, not vice-versa.

Finally, let us make a few irresistable observations. The whole notion of coupling via meaningless tokens is aesthetically repugnant and quite contrary to "pure" production system spirit. By "meaningless" we mean entries in DS that provide a narrow hand-crafted channel of communication between two specific rules that therefore "know about each other".[5] At the least, when a coupled rule deposits some "intermediate-state" message in a DS, one would like that message to be meaningful to many rules in the system, to have some significance itself. We can see that entries in a DS have an expected meaning to the read access functions that examine the DS.[6] If this purity is maintained, then any apparent "coupling" would be merely superficial: each rule could stand alone as a whole domain-dependent heuristic. Thus no harm should come from changing a single rule, and more rules could be added that act on the "intermediate message" of the coupling. Such meaningful, dynamic couplings should be encouraged. Only the meaningless, tight couplings are being criticized here.

4.3. Distribution of Knowledge Between Rules and DS

A common "pure" idea is that all knowledge of substance ought to be represented as rules. Independent of such rules, the DS forms no meaningful whole initially, nor has it any final interpretation. The "answer" which the RS computes is not stored in the DS; rather, the answer consists in the process of rule firings.[7] The DS is "just" an intermediate vehicle of control information.

Contrary to this, we say that rules ought to have a *symbiotic* relationship to DSs. The DSs hold meaningful domain-dependent information, and rules process knowledge represented in them. For RSs designed to perform scientific research, the DSs contain the theory, and the rules contain methods of theory formation.

But much domain-dependent knowledge is conditional. E.g., "If n and m are relatively prime and divide x, then so must nm". Shouldn't such If/Then information be encoded as rules? We answer an emphatic *No*. Just as there is a distribution of "all knowledge of substance" between rules and DSs, so too must the conditional information be partitioned between them. We shall illustrate two particular issues: (i) Much information can be stored implicitly in DSs; (ii) Some conditional knowledge is inappropriate to store as rules.

---

[5] By contrast, a "meaningful" DS entry will embody a piece of information which is specific to the RS's task, not to the actual rules themselves.

[6] Perhaps this "meaning" could even be expressed formally as an invariant which the write access functions for the DS must never violate.

[7] The sequence of actions in time. In addition, perhaps, the "answer" may involve a few of their side-effects. E.g., (Respond 'YES').

When designing a DS, it is possible to provide mechanisms for holding a vast amount of information *implicitly*. In AM, e.g., the organization of concepts into a Genl/Spec hierarchy (plus the assumed heritability properties; see 3.4) permits a rule to ask for "all concepts more general than Primes" as if that were a piece of data explicitly stored in a DS. In fact, only direct generalizations are stored ("The immediate generalization of Primes is Numbers"), and a "rippling" mechanism automatically runs up the Genl links to assemble a complete answer. Thus the number of specific answers the DS can provide is far greater than the number of individual items in the DS. True, these DS mechanisms will use up extra time in processing to obtain the answer; this is efficient since any particular request is very unlikely to be made. Just as each rule knows about a general situation, of which it will only see a few instances, that same quality (of wide potential applicability) is just as valuable for knowledge in DSs. These are situations where, like Dijkstra's multiplier [8], the mechanism must provide any of the consequences of its knowledge quickly on demand, but in its lifetime will only be asked a few of them.

Now that we have seen how tacit information *can* be encoded into DSs, let us see some cases where it *should* be -- i.e., where it is not appropriate to encode it as rules of the system. Many things get called implication, and only some of them correspond to rule application. For instance, there is logical entailment (e.g., if A∧B then A), physical causation (e.g., if it rains, then the ground will get wet), probable associations (e.g., if it is wet underfoot, then it has probably been raining.) These all describe the way the world is, not the way the perceiver of the world behaves. Contrast them with knowledge of the form "If it is raining, then open the umbrella". We claim that this last kind of situation-action relationship should be encoded as rules for the RS, but that the other types of implication should be stored declaratively within the DS. Let's try to justify this distinction.

The situation-action rules indicate imperatively how to behave in the world; the other types of implication merely indicate expected relationships and tendencies within the world. The rules of a RS are meant to indicate potential procedural actions which are obeyed by the system, while the DSs indicate the way the world (the RSs environment) behaves in terms of some model of it. The essential thing to consider is what relations are to be *caused in time*; these are the things we should cast as rules. The lhs of a rule measures some aspect of knowledge presently in DSs, while the rhs of the rule defines the attention of the system (regarded as a processor feeding off of the DS) in the immediate future.

This is the heart of why rule-sets are algorithms. They are algorithms for guiding the application of other (DS processing) algorithms. It also explains why other kinds of implications are unsuitable to be rules. Consider causal implication ("Raining --> Wet"). While the lhs could be a rule's lhs (it measures an aspect of any situation), the rhs should *not* be a rule's rhs (it does not indicate an appropriate action for the system to take).[8]

---

[8] In a RS that aspires to any generality at all, an antecedent theorem of the form "if [you know that] it is raining, then [assert that] it is wet" is not the appropriate form to store this knowledge; it is too compiled a form,

Most purist production systems have (often implicitly!) a rule of the form "If the left side of an implication is true in the database, Then assert the right side". This is only one kind of rule, of course, capable of dealing with implications. For example, MYCIN and LT [17] (implicitly) follow a very different rule: "If the rhs of an implication will satisfy my goal, Then the lhs of the implication is now the new goal". Other rules are possible; many rules for reasoning may feed off the same "table" of world knowledge. The point is that the implications themselves are declarative knowledge, not rules. In summary, then, it may be very important to distinguish rules (attention guides) from mere implications (access guides), and to store the latter within the DSs. This policy was not motivated by the scientific inference task for our RS. We believe it to be a worthwhile guideline in the design of any RS.

4.4. Interpreter

After a rule fires, the neo-classical interpretation policy (#9 in Figure 1) demands that *any* rule in the system can potentially be the next one selected to fire. This is true regardless of the speed-up techniques used in any particular implementation (say, by preprocessing the lhs's into a discrimination net [22]). But consider RSs for scientific discovery tasks. Their task -- both at the top level and frequently at lower levels -- is quite open-ended. If twenty rules trigger as relevant to such an open-ended activity (e.g., gathering empirical data, inducing conjectures, etc.) then there is much motivation for continuing to execute just these twenty rules for a while. They form an *ad hoc* plausible search algorithm for the agenda item selected.

A RS for discovery might reasonably be given a complex interpreter (rule-firing policy). AM, for example, experimented with a two-pass interpreter: first, a best-first, agenda-driven resource allocator and attention focusser selects the job it finds most interesting; second, it locates the set of relevant rules (typically about 30 to 40 rules) for the job, and begins executing them one after another (in best-first order of specificity) until the resources allocated in the first step run out [20]. The overall rating of the job which these rules are to satisfy determines the amount of cpu time and list cells that may be used up before the rules are interrupted and job is abandoned.

For example, say the job were "Find examples of Primes". It's allotted 35 cpu seconds and 300 list cells, due to its overall priority rating just before it was plucked from the agenda. Say, 24 rules are relevant. The first one quickly finds that "2" and "3" are primes. Should the job halt right then? No, not if the real reason for this job is to gather as much data as possible, data from which conjectures will be suggested and tested. In that case, many of the other 23 rules should be fired as well. They will produce not only *additional* examples, but perhaps other *types* of examples.

---

standing alone. If "told" (or given) a rule like this, a learning system should "parse" it as a familiar kind of deduction, file the residue of new information away as a conjectured tendency of wetness to follow rain, and start checking for exceptions. A sophisticated (and lucky) discovery RS might thereby develop the concept of "shelter".

The jobs on AM's agenda are really just mini-research questions which are plausible to spend time investigating. Although phrased as specific requests, each one is really a research proposal, a topic to concentrate upon. We found it necessary to deviate from the simplest uniform interpreter for clarity (e.g., a human can follow the first-pass (job selection) taken alone and can follow the second-pass (job execution) by itself), for efficiency (knowing that all 24 rules are relevant, there is no need to find them 35 times), and for power (applying qualitatively different kinds of rules yields various types of examples). We claim this quality of open-endedness will recur in any RS whose task is free concept exploration. This includes all scientific discovery but not all scientific inference.

5. Speculations for a New Discovery System

The spirit of this paper has been to give up straightforward simplicity in RSs for clarity, efficiency, and power. Several examples have been cited, but we speculate that there are further tradeoffs of this kind which are applicable to RSs whose purpose is to make new discoveries.

Often, there are several possible ways the designer may view the task of (and subtasks of) the intended RS. We wish to add the notion of "proof" to AM, say. Should we represent proof as a resolution search, as a process of criticism and improvement [11] spiralling toward a solution, as a natural deduction cascade, ...? Although any one of these task-views might perform respectably, we advocate the incorporation of all of them, despite the concommitant costs of added processing time, space, and interfacing. In fact, we wish never to exclude the possibility of the system acquiring another task-view.

We look for the development of further discovery tools in the form of domain-independent meta-heuristics that synthesize heuristic rules, and in the form of abstract heuristic schemata that specialize into efficient rules for each newly-discovered domain. These discovery tools are all part of "getting familiar" with shallowly understood concepts, such as synthesized ones tend to be initially. It may even be that symbolic analogy techniques exist, cutting across the traditional boundaries of knowledge domains.

We contemplate a system that keeps track of (and has methods with which it attempts to improve) the design of its own DSs, its own control structure, and perhaps even its own design constraints. Although working in (a collection of) specific domains, this would be a general *symbol system discoverer*, capable of picking up and exploring formulations, testing them and improving them.

5.1. A New Set of Design Constraints

Below are 13 principles for designing a RS whose task is that of scientific theory formation. They are the result of reconsidering the original principles (Figure 1) in the light shed by work on AM. Most of the "pure" principles we mentioned in Figure 1 are changed, and a few new ones have emerged.

FIGURE 3: Scientific Discovery RS Architecture

1. *Principle of Several Appropriate Memories. For each type of knowledge which must be dealt with in its own way, a separate DS should be maintained. The precise nature of each DS should be chosen so as to facilitate the access (read/write) operations which will be most commonly requested of it.*

2. *Principle of Maximal DS Accesses. The set of primitive DS access operations (i.e., the read tests which a rule's lhs may perform, and the write actions which a rhs may call for) are chosen to include the largest packages (clusters, chunks,...) of activity which are commonly needed and which can be performed efficiently on the DS.*

3. *Principle of Facetted DS Elements. For ever-growing data structures, there is much to be gained and little lost by permitting parts of one DS item to point to other DS items. In particular, schematic techniques of representing content by structure are now possible.*

4. *Principle of Rules as Data. The view which the RS designer takes of the system's task may require that some rules be capable of reasoning about the rules in the RS (adding new ones, deleting old ones, keeping track of rules' performance, modifying existing rules,...). Some of the methods the RS uses to deal with scientific knowledge may be applicable to dealing with rules as well. In such cases, the system's rules may thus be naturally represented as new entries in the existing DS which holds the scientific theory.*

5. *Principle of Regularities Among Rules. Each rule is actually a rule <u>schema</u>. Sophisticated processing may be needed both to determine which instance(s) are relevant and to find the precise sequence of actions to be executed. Such schemata are often quite elaborate.*

6. *Principle of Avoiding Meaninglessly-Coupled Rules. Passing special-purpose loop control notes back and forth is contrary to both the spirit of pure RSs and to efficiency. If rules are to behave as coupled, the least we demand is that the notes they write and read for each other be meaningful entries in DS (any other rule may interpret the same note, and other rules might have written one identical to it).*

7. *Principle of Controlled Environment. For many tasks, it is detrimental to permit external stimuli (from an environment) to enter any DS at random. At the least, the RS should be able to distinguish these alien inputs from internally-generated DS entries.*

8. *Principle of Tacit Knowledge. In designing the DS, much knowledge may be stored <u>implicitly</u>; e.g., by where facts are placed in a hierarchical network. The DS should be designed so as to maximize this kind of concentrated, analogical information storage. Hence, hard-working access functions are needed to encode and decode the full meaning of DSs.*

9. *Principle of Named Algorithms. When basic, "how to" knowledge is available, it should be packaged as an operation and used as a part of the lhs or rhs of various rules. Embodying this chunk of knowledge as several coupled rules is not recommended, for we will want to manipulate and utilize this knowledge as a whole.*

10. *Principle of Rules as Attention Guides. Knowledge should be encoded as rules when it is intended to serve as a guide of the system's attention; to direct its behavior. Other kinds of information, even if stated in conditional form, should be relegated to DSs (either explicitly as entries, or implicitly as special access functions).*

11. *Principle of Inertial Interpreter. In tasks like scientific research, where relevant rules will be performing inherently open-ended activities (e.g., data-gathering), such rules should be allowed to continue for a while even after they have nominally carried out the activity (e.g., gathered one piece of data). In such cases, the occasional wasted time and space is more than compensated for by the frequent acquisition of valuable knowledge that was concentrated in the later rules. For scientific discovery, no single rule (however "appropriate") should be taken as sufficient: a single rule must necessarily view the task in just one particular way. All views of the task have something to contribute; hence variety depends on a policy of always applying several rules.*

12. *Principle of Openness. A discovery rule system can be enriched by incorporating into its design several independent views of the knowledge it handles. Never assume everything is known about a class of knowledge. All appropriate formulations of a knowledge class have something to contribute; hence variety depends on openness to new formulations.*

13. *Principle of Support of Discovery by Design. By representing its own design explicitly (say, as concepts), the RS could study and improve those concepts, thereby improving itself. This includes the DS design[9], the access function algorithms, how to couple them, the function of various rules, the interpretation policy of the RS, etc. This suggests that the study of designs of computational mechanisms may be a worthy area for a discovery system to pursue, whether its own design is available to it or not.*

Rule systems whose designs adhere to these guidelines will be large, elaborate, and non-classical. We have mentioned throughout the paper several new complications which the principles introduce. Trying to produce such a RS for a task for which a

---

[9] e.g., the facet specifications. If the input/output requirements change with time, so should the rule system's data structures.

pure, neo-classical production rule system was appropriate will probably result in disaster. Nevertheless, empirical evidence suggests that RSs having this architecture are quite natural -- and relatively tractable to construct -- for open-ended tasks like scientific discovery.

ACKNOWLEDGEMENT

This research builds upon Lenat's Ph.D. thesis at Stanford University, and he wishes to deeply thank his advisers and committee members: Bruce Buchanan, Edward Feigenbaum, Cordell Green, Donald Knuth, and Allen Newell. In addition, he gladly acknowledges the ideas he received in discussions with Dan Bobrow, Avra Cohn, and Randy Davis. Similarly, ideas received by Harris in two long and fruitful associations, with John Seely Brown and with Roger Schank, have contributed to this work. Many of our ideas have evolved through discussions at CMU this past year, notably with Don Cohen, John McDermott, Kamesh Ramakrishna, Paul Rosenbloom, James Saxe, and especially Herbert Simon.

REFERENCES

[0] Bledsoe, W. W., and Tyson, M., *The UT Interactive Prover*, University of Texas at Austin, Depts. of Mathematics and Computer Sciences Automatic Theorem Proving Project Report No. 17, May, 1975.

[1] Bobrow, D., "Natural Language Input for a Computer Problem Solving System", in (Minsky, M., editor), *Semantic Information Processing*, The MIT Press, Cambridge, Massachusetts, 1968.

[2] Bobrow, D., and Winograd, T., *An Overview of KRL, A Knowledge Representation Language*, Journal of Cognitive Science, Vol 1, No 1, January 1977.

[3] Bobrow, R., and Brown, J. S., "Systematic Understanding, in (Bobrow, D., and Collins, A., eds.), *Representation and Understanding*, Academic Press, S.F., 1975.

[4] Buchanan, Bruce G., G. Sutherland, and E. Feigenbaum, *Heuristic Dendral: A Program for Generating Explanatory Hypotheses in Organic Chemistry*, in (Meltzer and Michie, eds.) Machine Intelligence 4, American Elsevier Pub., N.Y., 1969, pp. 209-254.

[5] Buchanan, Bruce G., *Applications of Artificial Intelligence to Scientific Reasoning*, Second USA-Japan Computer Conference, Tokyo, August 26-28. Published by AFIPS and IPSJ, Tokyo, 1975, pp. 189-194.

[6] Davis, Randall, *Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*, SAIL AIM-271, Artificial Intelligence Laboratory, Stanford University, July, 1976.

[7] Davis, R., and King, J., *An overview of production systems*, Report STAN-CS-75-524, Memo AIM-271, Stanford U. CS Department, 1975.

[8] Dijkstra, E. W., "Notes on Structured Programming", in Dahl, Dijkstra, and Hoare, *Structured Programming*, Academic Press, London, 1972, pp. 1-82.

[9] Hayes-Roth, Frederick, and Victor R. Lesser, *Focus of Attention in a Distributed*

Speech Understanding System, Computer Science Dept. Memo , Carnegie Mellon University, Pittsburgh, Pa., January 12, 1976.

[10] Hewitt, Carl, *Viewing Control Structures as Patterns of Passing Messages*, MIT AI Lab Working Paper 92, April, 1976.

[11] Lakatos, Imre, *Proofs and Refutations*, Cambridge U. Press, 1976.

[12] Lenat, D., *BEINGs: Knowledge as Interacting Experts*, 4th IJCAI, Tbilisi, Georgian SSR, USSR, 1975.

[13] Lenat, D., *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*, SAIL AIM-286, Artificial Intelligence Laboratory, Stanford University, July, 1976. Jointly issued as Computer Science Dept. Report No. STAN-CS-76-570.

[14] McCracken, Don, *A Parallel Production System Architecture for Speech Understanding*, CMU CS Dept. Ph.D. Thesis, 1977.

[15] Minsky, Marvin, "A Framework for Representing Knowledge", in (Winston, P., ed.), *The Psychology of Computer Vision*, McGraw Hill, N.Y. 1975, pp. 211-277.

[16] Moran, T.P., *The symbolic imagery hypothesis: An empirical investigation via a production system simulation of human behavior in a visualization task*, CMU CS Dept. Thesis, 1973. See also 3IJCAI, pp. 472-477.

[17] Newell, Allen, J. Shaw, and H. Simon, *Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristics*, RAND Corp. Report P-951, March, 1957.

[18] Newell, Allen, and Simon, Herbert, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[19] Newell, A., *Production Systems: Models of Control Structures*, May, 1973 CMU Report, also published in (W.G. Chase, ed.) *Visual Information Processing*, NY: Academic Press, Chapter 10, pp. 463-526.

[20] Norman, D., and D. Bobrow, *On Data-limited and Resource-limited Processes*, Journal of Cognitive Psychology, Volume 7, 1975, pp. 44-64.

[21] Polya, George, *Mathematics and Plausible Reasoning*, Princeton University Press, Princeton, Vol. 1, 1954; Vol. 2, 1954.

[22] Rychener, M. D., *Production systems as a programming language for artificial intelligence applications.* Pittsburgh, Pa: Carnegie-Mellon University, Department of Computer Science. 1976.

[23] Shortliffe, E. H., *MYCIN -- A rule-based computer program for advising physicians regarding antimicrobial therapy selection*, Stanford AI Memo 251, October, 1974.

[24] Waterman, D. A., *Adaptive Production Systems*, CIP Working Paper 285, CMU Psychology Dept., 1974. See also 4IJCAI, pp. 296-303.

[25] Waterman, D. A., *Generalization Learning Techniques for Automating the Learning of Heuristics*, AI Journal (forthcoming)