Report 82-31
Stanford -- KSL

Scientific DataLink

MRS/NEOMYCIN: Representing Metacontrol in Predicate Calculus.

William J. Clancey, Conrad Bock,

Jan 1983

Heuristic Programming Project
Report No. HPP-82-31

December 1982

MRS/NEOMYCIN:

Representing Metacontrol in Predicate Calculus

William J. Clancey and Conrad Bock
Stanford University
Heuristic Programming Project

January 1983
(revised)

MRS/NEOMYCIN:

Representing Metacontrol in Predicate Calculus

William J. Clancey and Conrad Bock

Stanford University

Heuristic Programming Project

January 1983

Table of Contents

1. Abstract 1
2. Introduction 1
3. Background 2
    - 3.1. Description of NEOMYCIN 3
        - 3.1.1. Architecture 3
        - 3.1.2. The Task Interpreter 4
    - 3.2. Description of MRS 5
        - 3.2.1. The Database 6
        - 3.2.2. Inference Procedures 6
        - 3.2.3. Procedural Attachment 7
4. Description of MRS/NEOMYCIN 7
    - 4.1. From NEOMYCIN and MRS to MRS/NEOMYCIN 7
    - 4.2. Deliberation/Action Loop 8
    - 4.3. Deducing the NEXTACTION 9
5. Sample Trace of MRS/NEOMYCIN 11
6. Summary of Techniques for Procedural Application of Rules 16
    - 6.1. Non-try-all Application of Rules 16
    - 6.2. Try-all (Sequential) Application of Rules 17
    - 6.3. Applying Rules Until They Fail 17
    - 6.4. Applying Rules Once 17
    - 6.5. Caching Results 18
    - 6.6. If-then-else Statements 18
7. MRS Augmentations 18
8. Problems and Successes of MRS/NEOMYCIN 20
    - 8.1. Procedural Application of Rules 20
    - 8.2. Deliberation/Action Loop 21
    - 8.3. Recomputation and Recording 22
    - 8.4. Procedural Attachment and Incremental Translation 22
    - 8.5. Efficiency 22
9. Conclusion 23
10. Acknowledgements 24

1. Abstract

This paper describes techniques for representing control knowledge in predicate calculus. A hybrid system is described in which metarules for diagnostic problem solving and their interpreter (both supplied by the NEOMYCIN program) are expressed in a form of predicate calculus (supplied by MRS). Procedural attachment is used to access and execute the untranslated domain knowledge.

A simple deliberation/action loop manages the system at the highest level. There are three metalevels of reasoning. Metarules recommend action in terms of domain rules to apply, questions to ask the user, or abstract tasks requiring application of more metarules. Metacontrol rules, corresponding to the interpreter for metarules, deduce what action to take next. The highest level of rules orders and selects the metacontrol rules, referring to them by name.

Sequential and iterative application of rules is accomplished by ordering rules in the database, keeping records of which rules have been applied or failed, and by using a form of backward chaining that dynamically calculates the rules to be used in deduction. This architecture and vocabulary of primitives provides a basis for improving NEOMYCIN's explanation and knowledge-acquistion capabilities, as well as increasing the usefulness of MRS for representing metacontrol.

2. Introduction

In the broadest description, MRS/NEOMYCIN[1] is a combination of:

- NEOMYCIN [3], a cognitive model of medical diagnosis, designed to be used in a program for teaching diagnostic strategies;

- MRS [7], a representation system for handling deduction and metacontrol;

- augmentations to MRS installed during the construction of MRS/NEOMYCIN.

More specifically, MRS/NEOMYCIN is NEOMYCIN with its metacontrol portion represented in the *augmented MRS*.

Metacontrol is defined here as ordering the application of domain rules and requests for data in a production system. *Domain rules*, or *base-level rules*, are the rules that refer only to the internal description of the real world. In NEOMYCIN these rules refer to the properties of a patient, for example, disease symptoms and social history. The metacontrol portion of a production system controls the use of domain rules.

---

[1] The design and implementation of MRS/NEOMYCIN is primarily the work of Conrad Bock, in partial fulfillment of the Master's degree in Artificial Intelligence at Stanford University. Funding was provided in part by ONR14-79C-0302 and SUMEX-AIM NIH RR-00785.

The representation of metacontrol knowledge has been studied by other researchers. Davis [4] considers the use of metarules for refining the invocation of base level rules. Georgeff [6] proposes a framework for procedural control of production systems. Genesereth [7] demonstrates how predicate calculus can be used to represent procedural control knowledge. The work reported here blends the purposes and methods of earlier research. In MRS/NEOMYCIN an interpreter for procedurally controlling the invocation of metarules, as well as the metarules themselves, are uniformly represented in predicate calculus. At the highest level, the system is managed by a simple *deliberation/action* loop. Together, the deliberation/action loop and metacontrol rules constitute the "inference engine."

NEOMYCIN and MRS are particularly suited for studying metacontrol. The purpose of NEOMYCIN is to develop a knowledge base that facilitates recognizing and explaining diagnostic strategies [1]. The approach has been to model human reasoning, representing control knowledge explicitly. By *explicit* we mean that the control knowledge is stated abstractly in rules, rather than embedded in application-specific code, and that the control rules are separate from the domain rules[2] In contrast to Davis's metarules, NEOMYCIN's metarules choose among lines of reasoning, as well as among individual productions.

MRS was designed with metacontrol research in mind, however, as Davis [4] points out, making representation of control possible is not the same as making it easy. MRS itself provides no vocabulary of control concepts or primitives for representing control. NEOMYCIN provides this framework--a hierarchically controlled production system as described by Georgeff--but its original representation uses arbitrary LISP code in metarule premises and the metarule interpreter.

MRS/NEOMYCIN extends the efforts of NEOMYCIN and MRS, translating NEOMYCIN's metacontrol code into rules and developing new metacontrol mechanisms for MRS. In addition, some problems and advantages of using MRS for system building are isolated. To be more specific than this requires background on NEOMYCIN and MRS.

3. Background

The descriptions of NEOMYCIN and MRS will be limited to those aspects relevant to MRS/NEOMYCIN. It is not claimed that these are the most important aspects with regard to the individual projects in general.

---

[2] See [2] for discussion of how diagnostic procedures can be captured by rules and still not be explicit.

3.1. Description of NEOMYCIN

3.1.1. Architecture

To achieve explicit control knowledge, many changes and additions were made to MYCIN. The additions of concern here are the *metarules*, the *tasks*, and the *task interpreter*.

The metarules control domain-rule application, data requests, and task invocation (described next). The premises of the metarules[3] select the domain rules to be applied and the questions to be asked. At any given point in a consultation, the decision about how to proceed is based on findings supplied to the program (e.g., temperature of the patient) and the properties of the problem solution (e.g., current diagnostic hypotheses).

Figure 3-1 shows a typical metarule, as represented originally in NEOMYCIN. This rule is tried in order to confirm that a disorder is present. It is a metarule because it has the effect of selecting domain rules. The domain rules selected conclude about the disorder being focused on and mention findings in their premise part that are causal pre-requisites for the disorder. This metarule is domain-independent in the sense that it mentions no disorders, findings, or rules that specific to a particular domain, and it makes sense to apply this metarule in different domains.

Note that the premise calls a LISP function and the action calls MYCIN's routine for applying domain-level rules. The premise function is opaque to the program, so it cannot explain why the rule fails nor precisely how the list of rules is calculated. This inhibits the student-modelling and knowledge-acquisition capabilities of the program. In MRS/NEOMYCIN, this function is expressed in MRS rules.

```
Premise: (SETQ RULELST (CAUSAL-PRE-REQUISITE? CURFOCUS))

Action:  (APPLYRULES RULELST)

Task: TEST-HYPOTHESIS

Task-when: DODURING
```

Figure 3-1: Typical NEOMYCIN metarule.

NEOMYCIN's metarules are controlled with some basic elements of traditional programming, but applied to metarule application instead of statement execution. The elements used are procedures, iteration, sequential execution, and nonlocal procedure exits. Also used is the

---

[3] Rules in the MYCIN family are pairs of LISP functions, called the premise and the action. In application of a rule, the premise is evaluated; if it returns non-NIL, it "succeeds" and the action is evaluated.

nontraditional construct of keeping a history of completed procedures and their arguments.

A NEOMYCIN *task* controls the application of metarules in the same way that a traditional procedure controls the execution of statements. *Invoking* a task is analogous to a traditional procedure call. Tasks are also termed *abstract actions* because they are groupings of the *primitive* actions of domain rule application and data requests. Since tasks are themselves actions, they have names and are invoked by the action part of the metarules along with the primitive actions. The mechanism that executes the tasks is called the *task interpreter*. NEOMYCIN begins by executing the top-level task, to carry on a consultation. The tasks then direct the application of metarules; when metarules succeed, actions are taken. See Figure 3-2.

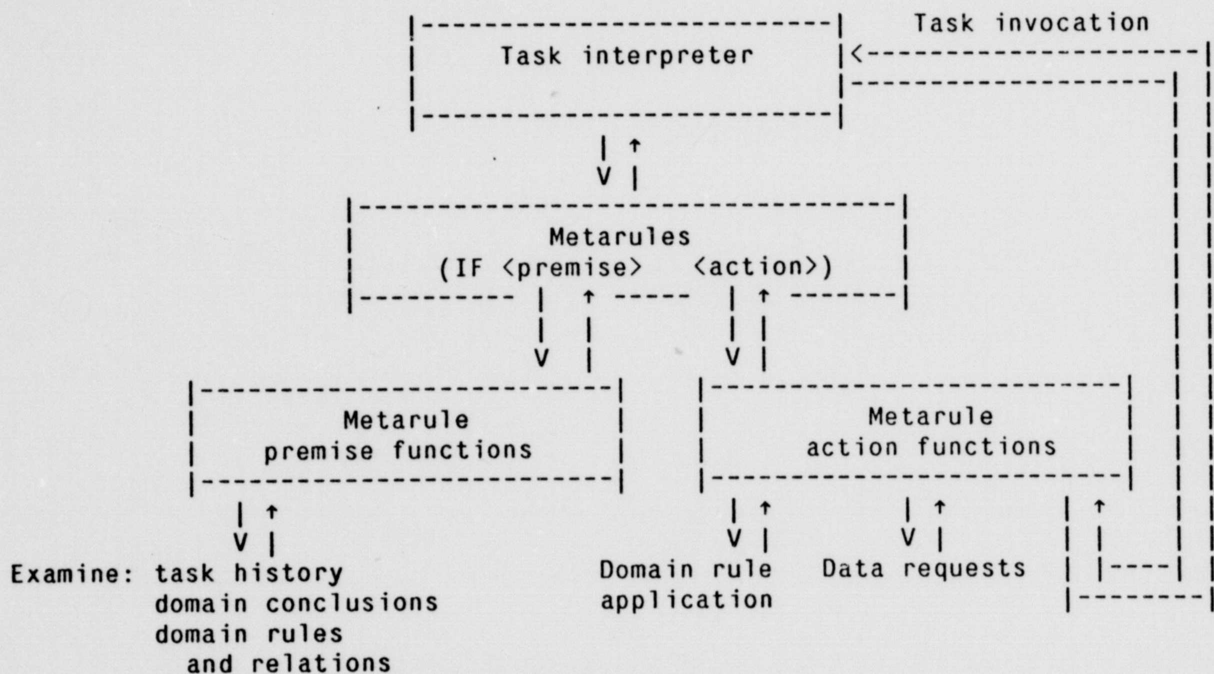

Figure 3-2: Neomycin control flow.

3.1.2. The Task Interpreter

The task interpreter executes a NEOMYCIN task by procedurally controlling how the metarules that compose it are applied. The basic elements of a task are:

- The *focus*, which is the argument of the task (e.g., the focus of the task TEST-HYPOTHESIS is the hypothesis to be tested).

- The *DODURING*, or *ACHIEVED-BY*, metarules, which are to be applied to complete the task. They are ordered.

- The *end condition*, which may abort the task *or any subtask* when it becomes true. Aborting can occur only while the DODURING metarules are being applied. The end condition is tested after each metarule of a task succeeds (see below).

- The *DOBEFORE* metarules, which are to be applied before the DODURING rules. They are ordered.

- The *DOAFTER* metarules, which are to be applied after the DODURING rules. They are ordered.

- The *goal*,[4] which is recorded to show that the task has been accomplished, unless the DODURING rules are aborted by the end condition. This *retained state information* is examined by the metarules.

- The *task type*, which specifies how the DODURING metarules are to be applied. There are two dimensions to the task type: *simple* or *iterative*, and *try-all* or *not-try-all*. The combinations give four ways of applying the DODURING rules:

  - *Simple, try-all.* The ruls are applied once each, in order. Each time a metarule succeeds, the end condition is tested.

  - *Simple, not-try-all.* The rules are applied in sequence until one succeeds, then the process stops.

  - *Iterative, try-all.* All the rules are applied in sequence. If there are one or more successes, the process is started over. The process stops when all the rules in the sequence fail. Each time a metarule succeeds, the end condition is tested.

  - *Iterative, not-try-all.* Same as for iterative try-all, except that the process is restarted after a single metarule succeeds.

MRS/NEOMYCIN's metacontrol rules use this information to execute a task.

3.2. Description of MRS

Although an attempt is made here to explain enough about MRS to understand MRS/NEOMYCIN, this is not intended to be an MRS tutorial. More detail is given in [8]. The aspects of MRS of concern here are the *database*, *inference procedures*, and *procedural attachment*.

---

[4] This use of *goal* here should not be confused with its use in backchaining.

3.2.1. The Database

The *database* is the only data structure in MRS. It is a collection of predicate calculus statements, such as

```
(COLOROF FIDO BROWN)
```

to indicate that Fido is brown, or

```
(AND (DOG FIDO) (COLOROF FIDO BROWN))
```

to indicate that Fido is a brown dog. The first element of a statement is called the *relation*, COLOROF, for instance, because it specifies the relationship among the objects that follow it in the statement. It is similar to the links in semantic nets. Frequently, a statement is referred to by its relation, for instance, a "COLOROF statement." To *assert* or *unassert* a statement means to add or remove it from the database, respectively.

Universal quantification is allowed and is represented with variables beginning with dollar signs. For example,

```
(IF (DOG $X) (BARKS $X)) ,
```

or, in long form,

```
(ALL X (IF (DOG X) (BARKS X))) ,
```

which means: "For all objects X in the universe, it is true that 'If X is a dog, then X barks.' " This is an example of a *rule*. A rule may be referred to by the relation of its conclusion. For example, the above rule is a "BARKS rule."

3.2.2. Inference Procedures

An *inference procedure* is a piece of LISP code that indicates whether a given statement is deducible from the database statements. The inference procedure of interest here, called *TRUEP*, performs its inference using *backward chaining*.

Suppose we have in the database the statements

```
(IF (DOG $X) (BARKS $X))
(IF (SPANIEL $X) (DOG $X))
(SPANIEL FIDO) .
```

Backward chaining allows deduction of the statement (BARKS FIDO), as follows. The universal variables of the first rule are instantiated to give

```
(IF (DOG FIDO) (BARKS FIDO)) .
```

Then, the premise of the instantiated rule is backchained. This leads to the instantiation of the second rule, which gives

```
(IF (SPANIEL FIDO) (DOG FIDO)) .
```

Since (SPANIEL FIDO) is in the database, (BARKS FIDO) can be deduced from the database.

3.2.3. Procedural Attachment

A central feature of MRS is that the name of the LISP function used to assert, unassert, or TRUEP a given statement can be deduced at run time. The name of the LISP function may be deduced from any of the information present in the database. When the function deduced is always the same for a given statement, this feature reduces to what is commonly known as *procedural attachment*. It is so named because a procedure is attached to a statement to specify the handling of that statement.

Though the name of the attached procedure may be deduced in any manner from the database, the most common way is to look on the database for a statement of the form

```
(TO <assert/unassert/truep> <statement> <function name>) .
```

These are called the *to* statements. They specify what LISP procedures should be used to assert, unassert, or TRUEP a given statement.

In the case of a TRUEP, the attached procedure is an inference procedure. For instance, suppose that during backchaining, one of the rules needed to test that M is an element of list L is:

```
(IF (MEMBEROF L M) <conclusion>) .
```

A special function MEMBEROF-TRUEP is specified as the procedural attachment by the statement:[5]

```
(TO TRUEP (MEMBEROF $LIST $MEMBER) MEMBEROF-TRUEP) .
```

So during the backchaining of the above rule, the MEMBEROF-TRUEP function is called to deduce the statement (MEMBEROF L M). MEMBEROF-TRUEP searches the list L for the member M.

4. Description of MRS/NEOMYCIN

The basic data structures and control flow of MRS/NEOMYCIN are described here. Section 7 gathers into one place the additions made to MRS that are described throughout the paper.

4.1. From NEOMYCIN and MRS to MRS/NEOMYCIN

MRS/NEOMYCIN was created from NEOMYCIN by changing the representation of the metacontrol portion. Specifically (refer to Figure 3-2):

- The metarules are translated from MYCIN's LISP-like format to an MRS's declarative format. Whereas rules in the MYCIN family are <premise function> - <action function> pairs, MRS rules are <premise statement> - <conclusion statement> pairs. They state that the conclusion statement is true if the premise statement can be shown to be true.

---

[5] Recall that the dollar signs indicate universal quantification, so this statement says, "to TRUEP *any* three-element statement that begins with MEMBEROF, use MEMBEROF-TRUEP."

To conform to the MRS rule format, the NEOMYCIN metarules, which evaluate their actions upon application, become, in MRS/NEOMYCIN, simply *recommendations for action*:

```
(IF <premise statement> (DODURING <action>)) .
```

This rule states that if the premise statement is true, then the statement (DODURING <action>) is true. The actions in MRS/NEOMYCIN are domain-rule application, data request, and task invocation.

- The code that controls the application of metarules, that is, the task interpreter, is translated into MRS rules. These rules are called the *metarule control rules*, or simply *metacontrol rules*. Here it was necessary to augment MRS, as explained later.

- Most of the functions that are used as predicates in the premises of the metarules are translated into MRS rules. They are called the *metarule premise rules*. These rules are applied during the backchaining of the metarules.

Left untouched are:

- Most of the NEOMYCIN data structures: hierarchies of domain rules and parameters. That is, the domain knowledge base is defined separately from the metacontrol knowledge. All links are via procedural attachment.

- Those functions too simple (arithmetic functions) or too complicated to translate. In MRS/NEOMYCIN, "too complicated" means the code is highly procedural or mixes deliberation and action too heavily (such as MYCIN's therapy-selection algorithm and rule-previewing mechanism).

- The functions used in the actions of the metarules (except for task invocation), namely, domain-rule application and data requests.

The parts of NEOMYCIN not changed in MRS/NEOMYCIN are collectively called the *NEOMYCIN groundwork*. Procedural attachment is used to interface the parts that are translated to the augmented MRS with the parts that are not.

4.2. Deliberation/Action Loop

With most of the NEOMYCIN control now expressed in rules, an interpreter is needed to execute them. This brings us to the heart of MRS/NEOMYCIN and the first MRS augmentation: the *deliberation/action loop* (DA loop, [5] [7]). As its name implies, it deliberates about the next action to take and then takes it. If the action is primitive (a domain-rule application or data request), it is passed to the NEOMYCIN groundwork to be executed. If the action is abstract (a task), the DA loop must

deliberate further to determine the next primitive action.[6] The function that performs actions is called DOACTION, defined in Figure 4-1. It is the action part of the deliberation/action loop.

As will be described below, the abstract actions (the tasks) require keeping a stack of tasks, as with traditional procedures. A task is performed with a call to the deliberation/action loop, because more deliberation is necessary to reduce an abstract action to primitive actions. The primitive actions are done with LISP function calls to the NEOMYCIN groundwork (two functions, in particular, APPLYRULES and FINDOUT).

```
(LAMBDA (ACTION)
  (if (TASK? ACTION) then (PUSH-TASK-STACK ACTION)
                          (DALOOP)
                          (POP-TASK-STACK)
                     else
                          (EVAL ACTION)))
```

Figure 4-1: Definition of DOACTION.

The definition of the deliberation/action loop appears in Figure 4-2. DALOOP deduces the next action to take and then passes it in a recursive call to DOACTION. The loop continues until it is unable to deduce what action to take and then quits. Since DALOOP is called from DOACTION, this means the task is finished and that the task stack is popped.

```
(LAMBDA NIL
  (loop.until.null
    (DOACTION (TRUEP '(NEXTACTION $ACTION)))))
```

Figure 4-2: Definition of DALOOP.

4.3. Deducing the NEXTACTION

In general, the deduction of the appropriate next action can be done by any inference procedure available in MRS. In MRS/NEOMYCIN, backward chaining is used. While the metarules are the rules that specify the next action in their conclusions, they are not backchained directly by the DA loop. The catch is that deciding which metarules are correct to backchain at any given time requires more deduction. This is because only the metarules associated with the current task are to

---

[6] The DA loop has its roots in the principle that deliberation on the course of action and the actual acting should occur separately. Deliberation is defined here as those computations that do not change the database; actions are defined as those that do. This definition is relative to the levels of domain, strategy, and database actions. For example, deliberating with diagnostic metarules requires that the result be written somewhere (an MRS action); similarly, a strategic diagnostic action leads to substantial domain-level deliberation.

be applied, and they are to be applied by a nontrivial procedure, the task interpreter described previously. The deduction of which metarules to backchain is done using the *metarule control rules* (MC rules). (See Figure 4-3. Ellipsis ("...") indicates that only parts of the rule premises are shown.)

However, there is a problem. The task interpreter, like each diagnostic task, is a procedure that itself must be controlled by rules: The rules corresponding to it, the metacontrol rules, cannot be simply backchained. The highest level rules actually conclude not what metarules to apply but which rules constituting the task interpreter (MC rules) to apply.

The form of backward chaining that deduces at run time which rules should be backchained will be called *RULEFOR* backward chaining.[7] It is one of the MRS augmentations and is distinguished from normal backward chaining wherein *all* rules concluding about the goal are backchained. RULEFOR backchaining decides what rules to use by deducing *RULEFOR statements*, namely,

```
(RULEFOR <statement> <rule>) ,
```

where <statement> is the statement being backchained, in this case (NEXTACTION <action>), and <rule> is the rule to be used.

In Figure 4-3 observe that there are two layers of RULEFOR deduction: first, to determine which task interpreter (MC) rules to apply, and second, to determine which metarules to apply. When metarules are applied, their premises are backchained. Statements in metarule premises are concluded by *metarule premise rules*. These are the rules that replace the LISP functions originally in NEOMYCIN's metarule premises. The backchaining stops when these premise rules access the data structures and functions in the NEOMYCIN groundwork.

To summarize, we have linearized the steps of an iterative procedure by referencing by name in rules that invoke them conditionally. MC rules correspond to steps of the task interpreter. Some steps must only be done once at the beginning, some are applied iteratively, others are applied once at the end. MC rules conclude what the NEXTACTION should be, for example, a DODURING action. The rules that conclude what MC rule to invoke are the NEXTACTION RULEFOR (NR) rules. Each NR rule refers to a single MC rule by name. The NR rules are an ordered set of rules that can be backchained normally.

The sample trace of MRS/NEOMYCIN in the following section gives examples of all kinds of rules (NR, MC, DR, metarule, metarule premise rule, domain rule) and shows their interaction.

---

[7] The name comes from "The RULE FOR <statement> is <rule>," meaning that <rule> is to be used in the backchain deduction of <statement>.

```
                                             (TRUEP
|------------------------------------|        '(NEXTACTION $action))
| NEXTACTION RULEFOR rules:          |  <--------------------|
|(IF ... <MC rule> ...               |                       |    Task
|    (RULEFOR (NEXTACTION $action)   |  ------------------|  |  invocation
|             <MC rule>))            |      <action>     |  |  |----------|
|------------------------------------|                   V  |  V          |
                                                   |------------------|   |
            |   ↑                                  | Deliberation/    |   |
            V   |                                  |    action loop   |   |
|------------------------------------|             |------------------|   |
| Metacontrol rules:                 |                   ↑  |  |          |
|(IF ... (DODURING $action ) ...     |                   |  |  |----------|
|    (NEXTACTION $action))           |                   |  |
|------------------------------------|                   |  |
                                                         |  |
            |   ↑                                        |  |
            V   |                                        |  |  Domain-rule
|------------------------------------|                   |  |   application
| DODURING RULEFOR rules:            |              Done |  |
|(IF ... (TASKRULE $metarule) ...    |                   |  |  or
|    (RULEFOR (DODURING $action)     |                   |  |
|             $metarule))            |                   |  |  data
|------------------------------------|                   |  |   request
                                                         |  |
            |   ↑                                        |  |
            V   |                                        |  |
|------------------------------------|                   |  |
| Metarules:                         |                   |  |
|(IF <premise> (DODURING <action>))  |                   |  |
|------------------------------------|                   |  |
                                                         |  |
            |   ↑                                        |  |
            V   |                                        |  |
      |-----------------------|                          |  |
      | Metarule premise rules |                         |  |
      |-----------------------|                          |  |
            |   ↑                                        |  |
            V   |                                        |  V
====================================NEOMYCIN groundwork====================
```

Figure 4-3: MRS/NEOMYCIN control flow (showing deduction of DODURING metarules).
[$x = variable; <x> = instantiation of $x]

5. Sample Trace of MRS/NEOMYCIN

Presented here is a trace from a MRS/NEOMYCIN run. Comments are in italics. The example uses a task called PROCESS-DATUM which does forward reasoning with new data. The statements describing it are given in Figure 5 (only 2 of 7 metarules are shown). The task is simple/try-all, so all of its DODURING metarules will be tried once. The task is invoked by some metarule which causes the DALOOP to execute:

```
(DOACTION '(PROCESS-DATUM NEUROSIGN-CHRONICITY))
```

```
(TASKTYPE PROCESS-DATUM SIMPLE)
(TASKGOAL PROCESS-DATUM PARTIALLY-PROCESSED)
(END-CONDITION PROCESS-DATUM DONT-ABORT)
(TASK-TRY-ALL PROCESS-DATUM)

(TASKRULE PROCESS-DATUM
          (IF (AND (CURRENT-ARGUMENT $FOCUS-DATUM)
                   (MKSET (CLARIFY-QUESTION $FOCUS-DATUM $CLARIFY-Q)
                          $CLARIFY-Q CLARIFY-Q-SET))
              (DODURING (FINDOUT CNTXT (MEMBERSOF 'CLARIFY-Q-SET)))))


(TASKRULE PROCESS-DATUM
          (IF (AND (CURRENT-ARGUMENT $FOCUS-DATUM)
                   (MKSET (RULES-IN-FOCUS? $FOCUS-DATUM $RULE)
                          $RULE RULE-SET))
              (DODURING (APPLYRULES (MEMBERSOF 'RULE-SET]
```

Figure 5-1: Some of the statements describing the task PROCESS-DATUM.

```
Starting abstract action PROCESS-DATUM on NEUROSIGN-CHRONICITY
```

The task PROCESS-DATUM is pushed onto the task stack. This is done with the relation CURRENT-TASK by asserting the statement

(CURRENT-TASK PROCESS-DATUM).

Similarly, NEUROSIGN-CHRONICITY is put on the CURRENT-ARGUMENT stack.

```
Starting DA loop
```

The DA loop executes

(TRUEP '(NEXTACTION $ACTION))

to find out what action should be passed to DOACTION. This invokes the RULEFOR backchainer which is attached to the NEXTACTION statements. The backchainer executes

(TRUEPS '(RULEFOR (NEXTACTION $ACTION) $RULE))

to find out what MC rules to use in deducing the NEXTACTION statement. (The plural form, TRUEPS, indicates that all of the rules that can be deduced from the database are returned.) The attached procedure for Normal backchaining is used to deduce the RULEFOR statements, which leads to backchaining the NEXTACTION RULEFOR (NR) rules. (Only NR rules 3 and 4 are shown below.)

For brevity, we will skip ahead to the third iteration of the DALOOP, after the MC rules intializing the task have completed and it is time to try the DOBEFORE rules.

This time around, all of the MC rules are candidates for concluding NEXTACTION, except for MC rules 1 and 2, which have already been applied (which is why NR rules 1 and 2 fail). Therefore, the deduction of the NEXTACTION statement begins with MC rule 3 (MC-3)...

```
Applying NR rule 1 ...
... failed
Applying NR rule 2 ...
... failed
```

```
Applying NR rule 3 ...

    [IF (AND (CURRENT-TASK $TASK)
             (THNOT (FAILED-IN-TASK $TASK MC-3]
        (RULEFOR (NEXTACTION $ACTION) MC-3]

... succeeded
Applying NR rule 4 ...

    [IF (AND (CURRENT-TASK $CURTASK)
             (OR [AND (OR (AND (TASKTYPE $CURTASK ITERATIVE)
                               (THNOT (TASK-TRY-ALL $CURTASK)))
                          (AND (TASKTYPE $CURTASK SIMPLE)
                               (TASK-TRY-ALL $CURTASK)))
                      (THNOT (FAILED-IN-TASK $CURTASK MC-4]
                 [AND (TASKTYPE $CURTASK SIMPLE)
                      (TASK-TRY-ALL $CURTASK)
                      (THNOT (APPLIED-IN-TASK $CURTASK MC-4]
        (RULEFOR (NEXTACTION $ACTION) MC-4]

... succeeded
Applying NR rule 5 ...
... succeeded
Applying NR rule 6 ...
... succeeded


Applying MC rule 3 ...

  (IF (DOBEFORE $ACTION)
      (NEXTACTION $ACTION))
```

The DOBEFORE, DODURING, DOAFTER statements have a RULEFOR backchainer attached. So rather than backchaining the DOBEFORE rules directly, we first backchain (RULEFOR (DOBEFORE $ACTION) $RULE) to determine which rules concluding (DOBEFORE $ACTION) should be tried. The rules concluding (RULEFOR (DOBEFORE $ACTION) $RULE) are the DOBEFORE RULEFOR (DR) rules. In this case, there is only one rule, DR rule 1.

```
Applying DR rule 1 ...

(IF (AND (CURRENT-TASK $CURTASK)
         (DOBEFORE-TASKRULE $CURTASK $MRULE)
         (THNOT (APPLIED-IN-TASK $CURTASK $MRULE)))
    (RULEFOR (DOBEFORE $ACTION) $MRULE))

... failed
```

DR rule 1 fails because the current task, PROCESS-DATUM, has no DOBEFORE rules associated with it (see description of PROCESS-DATUM at beginning of this section). That is, clause 2 of DR 1 cannot be deduced to be true.

... failed *(MC rule 3)*

MC rule 3 fails because its premise (DOBEFORE $ACTION) cannot be deduced. The deduction of the NEXTACTION statement continues with MC rule 4 ...

```
Applying MC rule 4 ...

  (IF (AND (DODURING $ACTION)
           (DONT-STOP-TASK PROCESS-DATUM))
      (NEXTACTION $ACTION))
```

RULEFOR backchaining is used to determine which metarules concluding DODURING should be tried...

```
Applying DR rule 2 ...

  (IF (AND (CURRENT-TASK $CURTASK)
           (TASKTYPE $CURTASK ITERATIVE)
           (TASKRULE $CURTASK $MRULE))
      (RULEFOR (DODURING $ACTION)
               $MRULE))

... failed
```

DR rule 2 fails because PROCESS-DATUM is not an iterative task.

```
Applying DR rule3 ...

  (IF (AND (CURRENT-TASK $CURTASK)
           (TASKTYPE $CURTASK SIMPLE)
           (TASKRULE $CURTASK $MRULE)
           (THNOT (APPLIED-IN-TASK $CURTASK $MRULE)))
      (RULEFOR (DODURING $ACTION)
               $MRULE))

... succeeded
```

DR rule 3 succeeds because PROCESS-DATUM is a simple task and has task rules that have not been applied. Since the RULEFOR backchainer uses TRUEPS on the RULEFOR statements, and all the DODURING metarules of PROCESS-DATUM have not been applied, all the metarules are returned as rules to use in deducing (DODURING $ACTION) for MC rule 4.

```
Applying  PROCESS-DATUM metarule 1 ...

  (IF (AND (CURRENT-ARGUMENT $FOCUS-DATUM)
           (MKSET (CLARIFY-QUESTION $FOCUS-DATUM $CLARIFY-Q)
                  $CLARIFY-Q CLARIFY-Q-SET))
      (DODURING (FINDOUT CNTXT (MEMBERSOF 'CLARIFY-Q-SET))))

... failed
```

This metarule uses the MRS augmentation MKSET (see section 7). Here the statement

(CLARIFY-QUESTION NEUROSIGN-CHRONICITY $CLARIFY-Q)

is TRUEPS'd, giving all the follow-up questions to ask about the datum NEUROSIGN-CHRONICITY.

CLARIFY-QUESTION is concluded by the metarule premise rule

(IF (AND (PROCESSQ $DATUM $QPARM)
(THNOT (TRACEDP CNTXT '$QPARM)))
(CLARIFY-QUESTION $DATUM $QPARM)).

Here procedural attachment is used. Recall that the domain knowledge of NEOMYCIN has not been translated to predicate calculus. The questions ($QPARM) that describe the process of a disorder (duration, severity, etc.) are stored as the PROCESSQ property of the disorder. Similarly, TRACEDP is a function that examines a property list to see if a question has been asked before.

If there are any questions, they are collected and put in the set called CLARIFY-Q-SET. Then FINDOUT asks the questions specified in the set. However, there are no questions in this case, so PROCESS-DATUM metarule 1 fails.

```
Applying PROCESS-DATUM metarule 2

  (IF (AND (CURRENT-ARGUMENT $FOCUS-DATUM)
           (MKSET (RULES-IN-FOCUS? $FOCUS-DATUM $RULE)
                  $RULE RULE-SET))
      (DODURING (APPLYRULES (MEMBERSOF 'RULE-SET))))

... succeeded

... succeeded (MC rule 4)
```

MC rule 4 succeeded partially because metarule 4 was able to deduce (DODURING $ACTION). Not shown is that the second clause of MC rule 4 needed to be deduced, namely

(DONT-STOP-TASK PROCESS-DATUM).

This is the clause that decides whether the task should be aborted or not. It is deduced by the following rule:

(IF (OR (END-CONDITION $CURTASK DONT-ABORT)
(ALLE (IF (AND (ONSTACK $TASK CURRENT-TASK)
(END-CONDITION $TASK $EC))
(THNOT $EC))))
(DONT-STOP-TASK $CURTASK))

The rule states that the end condition of all the tasks on the task stack must be undeducible, or the current task must have an end condition of DONT-ABORT. The latter is the case with PROCESS-DATUM.

Now that an MC rule has succeeded, DA loop passes the action to DOACTION. The action consists of applying the domain rules that MKSET put in the set RULE-SET, namely RULE407 and RULE403. These are rules that use the new datum NEUROSIGN-CHRONICITY to make conclusions about hypotheses that are currently "in focus."

```
Starting primitive action (APPLYRULES (MEMBERSOF 'RULE-SET))

Applying RULE407/PATIENT-2; RULE407 succeeded.
Conclude: MENINGITIS of PATIENT-2 is BACTERIAL-MENINGITIS (-.15)
Conclude: MENINGITIS of PATIENT-2 is VIRUS (-.09)
Conclude: CHRONIC-MENINGITIS of PATIENT-2 is YES (.24)

Applying RULE403/PATIENT-2; RULE403 succeeded.
Conclude: MENINGITIS of PATIENT-2 is BACTERIAL-MENINGITIS (.2)
Conclude: MENINGITIS of PATIENT-2 is VIRUS (.604)

Finished primitive action (APPLYRULES (MEMBERSOF 'RULE-SET))
```

After doing a single DODURING action, control returns to the DA loop, and the NEXTACTION statement is deduced again. NR rule 4 will again conclude that MC rule 4 should be tried--this is a try-all task so more than one metarule is allowed to succeed. MC rule 4 fails because it is unable to deduce (DODURING $ACTION)--there are no unapplied metarules. (Specifically, MC rule 4 fails because DR rule 3 fails to find any metarules that can be tried to conclude DODURING.)

The remaining NR rules will be tried, causing the DOAFTER metarules to be applied and goal of the task to be set. (Note that NR rule 4 will fail on subsequent iterations because MC rule 4 has failed.) When the task completes, DOACTION will pop the task and argument stack and erase the APPLIED-IN-TASK and FAILED-IN-TASK statements pertaining to PROCESS-DATUM.

6. Summary of Techniques for Procedural Application of Rules

The purpose of the metarule control rules is to mimic the task interpreter, the piece of the LISP code which applies the metarules in NEOMYCIN. The task interpreter is nicely expressed in a procedural language like LISP; it contains iterative application of rules, sequential application of rules, setting of variables to save results of computation, and if-then-else statements. These are difficult to express with rules. General examples below review how these constructs are handled in MRS/NEOMYCIN.

6.1. Non-try-all Application of Rules

The simplest form of rule application used in the NEOMYCIN task interpreter is that of a "pure production system" in which the first applicable rule[8] in the database is used every time backward chaining is done on the same statement. For instance, suppose the database contains the rules

```
(IF P1 G)
(IF P2 G)
(IF P3 G) .
```

At each call to the backchainer, these rules are tried in order,[9] starting at the beginning, until one succeeds. This technique implements iterative non-try-all application of rules. If controlled to occur only once, it will implement simple non-try-all tasks. Avoiding unnecessary computation when all of the rules are failing is discussed below.

---

[8] "Applicable rules" has different meanings for normal and RULEFOR backward chaining. For normal backchaining it means all the rules in the database that conclude about the statement being deduced. In RULEFOR backchaining it means the rules deduced at runtime as being applicable. If the same rule is deduced as being applicable at each call to the RULEFOR backchainer, then the same looping effect is achieved as with normal backchaining.

[9] See ordered database access entry in section 7 for how statements are ordered by the system designer.

6.2. Try-all (Sequential) Application of Rules

To apply rules sequentially, RULEFOR backward chaining is used. The technique is to mark that rules that have been applied already so that the RULEFOR backchainer will apply the next one in order.

For example, suppose that G is the goal. Let the database contain the rule

```
(IF (AND (RULE-IN-SEQUENCE $RULE)
         (NOT (APPLIED $RULE)))
    (RULEFOR G $RULE)) .
```

This rule states "for all objects $RULE, if $RULE is a rule in the sequence and has not been applied, then $RULE can be used to deduce G." Each time the RULEFOR backchainer is called, the first unapplied rule is backchained. Note that this requires the RULE-IN-SEQUENCE relation to find the rules in the order required. The APPLIED statements must be asserted into the database by the RULEFOR backchainer, since it is the only mechanism that knows when a rule is applied.

Note that sequential execution is different from iterative execution. The two are usually thought of together as the common DO loop (corresponding to iterative try-all tasks). In MRS/NEOMYCIN, iteration is brought about by a combination of the "loop.until.null" in DALOOP and the control rules that apply metarules until DODURING cannot be deduced.

6.3. Applying Rules Until They Fail

To avoid unnecessary recomputation, it is useful to avoid trying rules that cannot succeed after they have failed once. Again, RULEFOR backward chaining is used. Suppose that G is the goal and that POSSIBLE-RULE specifies the rules that might be used to deduce G. Let the database contain the rule

```
(IF (AND (POSSIBLE-RULE $RULE)
         (NOT (FAILED $RULE)))
    (RULEFOR G $RULE)) .
```

When G is being deduced, this rule will make the RULEFOR backchainer omit rules that have failed. As with the APPLIED statements used in sequential application, the FAILED statements must be recorded by the RULEFOR backchainer.

6.4. Applying Rules Once

Applying rules only once is straightforward. Using the previous example:

```
(IF (AND (POSSIBLE-RULE $RULE)
         (NOT (APPLIED $RULE)))
    (RULEFOR B $RULE)) .
```

Applying rules in sequence is a special case of this.

6.5. Caching Results

One way to avoid recomputation is to save the results of deduction. For example, consider the rule

```
(IF (AND A B) C) .
```

If the truth of A is important and does not change often, the result of trying to deduce A should be saved for efficiency. The solution in MRS/NEOMYCIN is to modify MRS's principle deductive function, TRUEP, to assert the result of the deducing statements the user wants cached. Then rules applied later can use the result.

For example, suppose that A has been indicated by the user as a statement to be cached. Then the application of the above rule would record the statement (CACHE A <result>) in the database. Another rule could refer to this later, for instance,

```
(IF (AND D (CACHE A T))
    E) .
```

For this rule to succeed, the previously deduced value of A must be T.

6.6. If-then-else Statements

MRS rules by their definition do not have "else" clauses, which are a method of avoiding recomputation. Caching technique is used to overcome the problem. First, the premise part of the IF is indicated for caching by the user, then the result can be referred to in another rule. For example,

```
(IF-THEN-ELSE A B C)
```

becomes

```
(IF A B)
(IF (NOT (CACHE A T)) C) ,
```

where the rules must be ordered, with no intervening rules that might change the result of deducing A.

7. MRS Augmentations

The MRS augmentations that were made in the course of building MRS/NEOMYCIN are summarized here:

- *The deliberation/action loop.* This is the "interpreter" of MRS/NEOMYCIN.

- *RULEFOR backward chaining.* This form of backchaining is used by the DALOOP to select dynamically the metacontrol and metarules to be used in deducing the next appropriate action.

- *Caching.* Caching is used to avoid unnecessary recomputation, such as for if-

then-else statements.

• *APPLIED and FAILED marking mechanisms.* These are used to bring about sequential and iterative rule application within a task. When the task is done, all the APPLIED and FAILED statements are erased. This is done so that the statements cannot confuse the operations of later calls to the task. Note that these statements would need to be included on the stack to allow tasks to be invoked recursively.

• *Ordered database access.* This is used to order the metarules. Since the MRS database in principle has no order imposed on the statements, the order must be specified in another set of statements. For instance, asserting the following statements would order statements F1, F2, and F3:

```
(ACCESSBEFORE (F1 A B) (F2 A B))
(ACCESSBEFORE (F2 A B) (F3 A B))
```

The inconvenience the ordering statements cause the programmer can be eliminated with a programmer interface function. The statements above are asserted as follows:

```
(ASSERT '(ORD* (F1 A B) (F2 A B) (F3 A B))) .
```

The function attached to ORD* statements asserts the statements shown along with their ordering relations. Also, a special procedural attachment is needed for TRUEP that sorts the original statements according to the ordering rules.

This method is too slow to use. A less elegant method is used in MRS/NEOMYCIN which employs the LIFO property of the MRS database. The ordered relations are put into the database in reverse order, so that TRUEP will find them in the order specified by the ORD* statement.

• *Extensional ALL (ALLE).* This construct allows statements like "All cows I know about are not purple." This is different from "All cows are not purple." The second requires some proof showing that cows, by their nature, cannot be purple. The first merely requires a database check.

Our cow example would give:

```
(ALLE (IF (COW $X) (NOT (PURPLE $X)))) .
```

In general, it allows statements of the form:

```
(ALLE (IF <premise> <conclusion>)) .
```

If the ALLE statement is true, it means that for all objects satisfying <premise>, <conclusion> is true. In MRS/NEOMYCIN, ALLE is used to check that none of the tasks on the task stack has its end condition satisfied.

• *Lambda statements.* NEOMYCIN's task structure requires a stack of tasks, analogous to traditional procedure-call stacks. That is, when a new task is invoked, the old task name and arguments are saved so that control can be returned to it when the new task is finished. Traditionally, this is called "lambda binding." In MRS/NEOMYCIN,

this is implemented by procedural attachment (to CURRENT-TASK, CURRENT-ARGUMENT, and ONSTACK statements).

• *Set collection (MKSET).* Frequently it is necessary to collect all the objects in the database that fulfill a given requirement. The solution used bends the rule about separation of deliberation and action. A relation MKSET is defined with the format

```
(MKSET <statement> <collectvar> <setname>) ,
```

where <collectvar> is a global variable. A procedure is attached so that when a MKSET statement is TRUEP'd, <statement> is TRUEPS'd and the bindings for <collectvar> are collected. The bindings are asserted as members of the set named <setname>. They can be retrieved with the function MEMBERSOF. The mechanism is used in the following metarule for applying domain rules that trigger hypotheses suggested by a recent datum given to the program:

```
(IF (MKSET (TRIGGERS? $FOCUSDATUM $TRIGRULE)
          $TRIGRULE TRIGRULESET)
    (DODURING (APPLYRULES CNTXT (MEMBERSOF 'TRIGRULESET))))
```

8. Problems and Successes of MRS/NEOMYCIN

This section discusses the advantages and disadvantages of the solutions found to various problems encountered in building MRS/NEOMYCIN. In the process, unresolved issues and future work are identified.

8.1. Procedural Application of Rules

Many times in MRS/NEOMYCIN, it is necessary to apply rules in a procedural manner, for example, iteration and if-then-else. One of the central issues in achieving this is machine and human readability of the representation. On one hand, we want a representation that allows the machine to reason about rule application; on the other, we want a representation that is easy for the programmer to read. It's not hard filling each of these requirements separately, but they seem to work against each other when brought together.

The representation used in MRS/NEOMYCIN was developed to allow machine readability, specifically, to allow the augmented MRS reasoning mechanisms to deduce what metarules should be applied at what time. While this was achieved, we must not confuse readability with understanding. Predicate calculus is often proposed as a means of making knowledge explicit, but just *stating* the steps of the procedure in rules doesn't make it understandable. (Consider the difficulty of understanding that NR rule 4 and DR rule 3 bring about simple try-all application of metarules.) The abstract properties of the program's *output* are not immediately obvious from the primitive terms and their combination in rules.

So it seems that, for the advantages of a simple interpreter (the DA loop) and the flexibility of having a program reason about control (allowing for dynamic changes in metacontrol), we have paid a considerable price in perspicuity. The original LISP code of the task interpreter, with abstract concepts such as "repeat-until," "first," and "finally," is more readable to a programmer. This same criticism can be made about the statement of control regimes given by Genesereth in [7]. MRS/NEOMYCIN is in no better position to explain what it is doing than the original LISP-coded program is. Fine-grained "explanation" has been made possible, but much more needs to be done before the program can be said to understand its procedures in an abstract sense.

A solution to the problem of human readability is to construct a compiler. In traditional programming languages, compilers are used to make assembly languages both machine-independent and easier to use for the programmer. In the same way, a procedural rule application compiler would provide a human interface and independence from the details of the MRS deductive mechanisms.

The problems with representation do not arise in the largely nonprocedural metarule premise rules. MRS allows matching constraints to be stated separately from the operations of search and database lookup, making these rules more programmer-readable than the original LISP code.

8.2. Deliberation/Action Loop

The problem has always come up in rule-based systems: Where should action be allowed to happen? With the DA loop, a framework for action is created. Separating deduction of the appropriate action and action itself makes database changes easier to see for both the programmer and explanation programs. Each action has a well-defined reason associated with it. Each deliberation leads to some action.

As with any restricted framework, however, some things are more difficult with the DA loop than without it. Not surprisingly, the difficulties occur in the same way they did with rules: representing procedural actions. An example from MRS/NEOMYCIN is when deliberations conclude that a *sequence* of actions should be done. The correct way to do this requires some deliberation in between each action that merely concludes to continue the sequence of actions previously specified. When there is no reason to stop the sequence, this is a waste of metareasoning. The solution adopted in MRS/NEOMYCIN is to sidestep the DA loop by creating a primitive action in the NEOMYCIN groundwork that executes the actions specified. A metarule concluding a sequence of actions has the form

```
(IF <premise> (DODURING (DOALL <action> <action> ...))) .
```

8.3. Recomputation and Recording

A problem directly related to separating deliberation and action is that no mechanism is provided for recording results of computation. The solutions given here basically sidestep the DA loop by allowing database changes during deduction (through caching and marking of rule application).

8.4. Procedural Attachment and Incremental Translation

In the early stages of MRS/NEOMYCIN, it was clear that time would not permit a complete translation of NEOMYCIN to the augmented MRS, but how much would need to be left incomplete was unknown. Fortunately, procedural attachment can be used to proceed incrementally. The translation of NEOMYCIN to rules can bypass parts of NEOMYCIN by attaching procedures that execute these untranslated parts (the NEOMYCIN groundwork). The resulting system is a hybrid (as reported by Rich [10]), allowing us to continue to use existing explanation and knowledge-acquisition packages for the groundwork, while developing new capabilities using the new representation at the strategical level.

8.5. Efficiency

The flexibility gained in the new representation of NEOMYCIN is paid for in efficiency. What was originally an 8-minute consultation became an 80-minute consultation. In the future, a faster version of LISP will be used on the Xerox Dolphin on which MRS/NEOMYCIN is executed. Also, a faster version of MRS will be used. These will help considerably, but the final results are unclear. The slowness of MRS/NEOMYCIN is due to:

- *The indirection inherent in retrieving procedural attachments.* Although an MRS shortcut allowed the attachments to be put on the LISP property lists of the statement relations, it is much slower than a normal LISP function call. This inefficiency is inherent in the architecture of MRS--no compiler could get rid of the indirection in TRUEPing during backchaining.

- *Unnecessary recomputation due to absence of certain pieces of metaknowledge.* For instance, in a simple try-all task all of the DODURING metarules of the task are applied once in order. However, MRS/NEOMYCIN does not know this explicitly. It re-deduces the DODURING metarules of the task, because it does not know that they are not different from the last time the task was invoked. Then, after each metarule that succeeds in providing a next action for the DA loop, the metarules are re-deduced leaving out the ones that have been applied already. This painful way of achieving sequential application is necessary because MRS/NEOMYCIN is unaware that it need not re-deduce the metarules in the sequence each time around the DA loop. It also does not

know that a list representation should be used for grouping the rules on the database.

A compiler that can reduce the procedural specifications is one answer to these problems. With the human interface compiler previously mentioned, there would be a compiler on the specification and execution sides of the procedural knowledge in MRS/NEOMYCIN. We have designed, but not implemented, a compiler that would essentially work by instantiating the NR and DR rules in the context of each task (simplifying, composing, and expanding them), allowing all metalevel deduction to be done by normal backchaining, without the RULEFOR levels of indirection.

9. Conclusion

The primary achievement of MRS/NEOMYCIN is representing in rules the procedural application of other rules, specifically translating NEOMYCIN's task interpreter and metarule premises into rules. This was done in the hope that the new representation will lead to better explanation capability for a program that teaches NEOMYCIN's expertise, as well as enhancing the debugging capability for knowledge acquisition. To specify procedural rule application in rules, MRS was augmented with a form of backward chaining (RULEFOR) that dynamically calculates rules to be used in deduction. Methods were found to allow RULEFOR backward chaining to guide the procedural application of rules. Similarly important is the implementation of a deliberation/action loop that can, in principle, remove database changes and user interaction from ad hoc procedural attachments on the rules. In addition, MRS augmentations were created to cope with recomputation inefficiencies.

The metaknowledge and architecture of MRS/NEOMYCIN that we have found to be useful for representing metacontrol in predicate calculus are summarized below.

- *Domain-independent metarules* for achieving strategical tasks: invoke base-level rules, invoke other tasks, and ask the user.

- *Structural knowledge* to relate base-level knowledge and the history of base-level and task knowledge application (not discussed in this paper; see [9]).

- *Task-specific control knowledge* for controlling metarules (the focus, type of task, goal, and end condition).

- *Metacontrol rules* (MC and DR rules), the task interpreter that applies metarules for a task, detects the end condition, and does bookkeeping.

- *Stack of tasks and focus arguments* so that the task interpreter can be invoked recursively (tasks can invoke other tasks).

- *History* of metarules and metacontrol rules applied or failed in a task for bringing about sequential and iterative computation.

- *Metametacontrol rules* (the NR rules) that refer to the metacontrol rules by name and invoke them in the proper sequence, allowing a normal backchaining interpreter to be used at the highest level.

As can be seen, we have constructed a fairly complex framework for reasoning about control, greatly elaborating upon Davis's original idea of using metarules for refining search [4]. In particular, domain-independent metarules invoke base-level rules, MC and DR rules order and choose metarules, and NR rules control the ordering process. This framework, a re-representation of the NEOMYCIN task interpreter, provides a vocabulary for control within MRS in terms of a *tasking mechanism*. This architecture and its primitives significantly increase the usefulness of MRS for representing metacontrol.

The solutions in MRS/NEOMYCIN create some new problems. The procedural specifications are cumbersome: They are hard to understand and are inefficient. One answer is to develop compilers. One compiler would provide a human interface to allow more abstract procedural specification; another compiler would reduce the procedural specifications to representations more efficient for the machine.

10. Acknowledgements

We thank Avron Barr, Greg Cooper, Lee Erman, and Diane Kanerva for commenting on an earlier version of this paper.

References

[1] W. J. Clancey.
Methodology for Building an Intelligent Tutoring System.
1981.
To appear in *Problems of Methodology in Cognitive Science*, Kintsch (Ed.).

[2] W. J. Clancey.
The Epistemology of a Rule-Based Expert System -- A Framework for Explanation.
Artificial Intelligence (in press), 1983.

[3] W. J. Clancey and R. Letsinger.
NEOMYCIN: Reconfiguring a Rule-Based Expert System for Application to Teaching.
In *Proceedings of the Seventh IJCAI*, pages 829-836. IJCAI, 1981.

[4] R. Davis.
Meta-Rules: Reasoning About Control.
Artificial Intelligence (15):179-222, 1980.

[5] J. Doyle.
A Model for Deliberation, Action and Introspection.
Technical Report 581, M.I.T. Artificial Intelligence Laboratory, 1980.

[6] M. P. Georgeff.
Procedural Control in Production Systems.
Artificial Intelligence (18):175-201, 1982.

[7] M. R. Genesereth and D. E. Smith.
Meta-Level Architecture.
MEMO HPP-81-6, Stanford University, 1982.

[8] M. R. Genesereth, R. Greiner, and D. E. Smith.
MRS Dictionary.
MEMO HPP-82-24, Stanford University, 1982.

[9] W. J. Clancey.
NEOMYCIN I: Tasks and Metarules for Diagnosis.
In preparation , Stanford University, 1983.

[10] C. Rich.
Knowledge Representation Languages and Predicate Calculus.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 193-196. AAAI, 1982.

Copyright © 1985 by KSL and
Comtex Scientific Corporation

FILMED FROM BEST AVAILABLE COPY