

VOCABULARIES FOR PROBLEM SOLVER STATE DESCRIPTIONS

Drew McDermott
Computer Science Department
Yale University
New Haven, Connecticut 06520

ABSTRACT: A model of knowledge representation is presented and applied to representing problem solvers' states. The focus is on the NASL problem solver, which has been used to study elementary circuit design. The model distinguishes between the form and the content ("vocabulary") of a representation. The vocabularies used by the modules of NASL are displayed. It is concluded that the model allows flexible implementation and clear description of problem solvers. In particular, it demonstrates the importance of shallow reasoning in the control of problem solvers.

Descriptive terms: Problem solvers, knowledge representation, computer-aided design, production systems, rule-based systems.

Acknowledgements: I thank Gerald Sussman for the word "vocabulary," and many of the concepts behind it. Much of the work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense, and monitored by the Office of Naval Research under contract number N00014-75-C-0643.

I Introduction

This paper is about the "representation of knowledge" in problem-solving systems. I focus on the representation of facts about the state of the problem solver itself, rather than on its representations of the current problem state. The discussion uses as an example the NASL problem-solving system. (McDermott, 1976) It is argued that the range of "representational vocabularies" determines the power of a problem solver.

Every representational system has two aspects: form and content. Its form determines what can be inferred from what; its content determines how what is inferred is used by the user of the representational system. In abstract form, such a system can be represented by Fig. 1.

Data Base <----> Gatekeeper <----> User Programs

Figure 1 General Representation System

The data base is managed by a "gatekeeper." Additions to the data base and requests to the gatekeeper from the user's programs are in terms

of the user's concerns. Inside the data base, they are translated into property-list operations, pattern matching, network marking, or some other internal representation for inferential processing.

Ideally, the formal conventions used inside the data base should be hidden from the user. There is a certain level of representational power and retrieval efficiency that he can take for granted (Moore, 1975), so that he will be free to think in problem-oriented terms. Another way to put this is that content must be allowed to be more important than form.

In this model, content appears as the vocabulary the user's programs use in talking to the gatekeeper. Since my focus in this paper is on the vocabulary, let me expand Fig. 1 to illustrate some points.



Figure 2 Structure of User's Programs

The user's programs often come in "modules," each of which is the primary holder of a specialized vocabulary. For example, in the MYCIN system (Shortliffe, 1976; Davis, 1976), there are two vocabularies: a system of medical terms and a "meta-vocabulary" used in controlling rule application. Inside the data base, most of the rules are in medical terms; the rule-application vocabulary is used by "meta-rules."

An alternative way of approaching these issues would be to let each module have its own data base, gatekeeper, and formal conventions, not just its own vocabulary. For example, in NOAH (Sacerdoti, 1975), the procedural net is stored separately (as special-purpose list structures) from the QLISP data base used to represent the current problem model. The problem with this approach is that it makes the communication channel between any two modules ad hoc and clumsy. In my model, the data base is the channel; any two rules in the data base can interact, regardless of their origin or principal vocabulary.

A consequence of my model is that symbols used by user programs have two kinds of "meaning": their "inferential" meaning, determined solely by the behavior of the data-base expressions in which they appear, and their "pragmatic" meaning, which depends on the way in which they alter the behavior of the user programs which employ them. Further, a user module can use a symbol in two different ways: in expressions added to the data base to trigger inferences (thus, as it were, telling itself what it's

doing), and in requests to the gatekeeper (which are often asking what to do next). Clearly, the same symbol may be used in both these ways.

II The NASL Problem Solver

NASL is a problem-reduction problem solver which has been applied mainly to the problem of designing simple electronic circuits. (McDermott, 1976) As will be seen shortly, it is implemented as interleaved planning and execution modules. The data base is managed by a PLANNER-like (Hewitt, 1972) predicate-calculus theorem prover, about which I will say no more here. (See McDermott, 1977.) Its "problem vocabulary" includes terms pertaining to devices, circuits, signals, and component values. What I will describe are its "control vocabularies."

A problem solver is a system which solves problems by reducing them to simpler subproblems, and ultimately to "primitives" which can be solved by built-in programs. A problem solver must maintain somewhere a data structure representing important features of its current state. Usually this data structure is different from the data structure representing the current problem state, or "world model." NASL, however, follows the structure of Figure 2, and maintains all its data structures in the same predicate-calculus data base.

This means that the formal system used by NASL restricts as little as possible the content and interaction of the problem-state and problem solver-state descriptions. In what follows, I will first describe the vocabularies used by NASL modules, then show rules which illustrate the interactions of the concepts they define.

NASL looks like this:

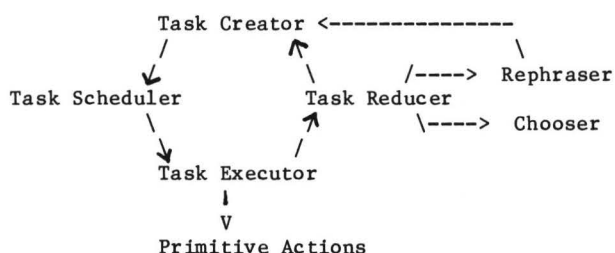


Figure 3 -- NASL Modules

These modules do not correspond one-to-one with programs, but are only conceptual. In Figure 3, an arrow from module A to module B means that A "depends on," "precedes," or "calls" B. The central cycle in the graph is intended to suggest that a problem, or "task," is created, then scheduled, then executed, either as a primitive action or by being reduced to subtasks.

As will be seen shortly, the notations used by these modules result in a task network being represented in the data base.

Two of the modules in Figure 3 are worthy of further description now. The task reducer tries to retrieve exactly one "plan" (set of subtasks) to carry out its task. This attempt can fail in two ways: either too many plans (more than one) are retrieved, or too few (none at all). In the first case, the choice module tries to choose among them. In the second, the rephrasing module creates a new task which treats the recalcitrant task as an object to be transformed into subtasks; that is, it brings all the problem solver's resources to bear on it.

II.A NASL's Vocabularies and their Pragmatic Meanings

Task Creation and Classification--

The basic predicate for describing tasks is

```
(TASK name
  < -input-data- > action < -output-data- >)
```

which defines a task with a certain name, which carries out a certain action. Tasks in NASL can deduce or create information, so there must be some mechanism for passing data from task to task.

For example, this formula

```
(TASK (COUPLER PLAN71)
  <(BUFFER72) (AMP73)>
  (LAMBDA (STAGE1 STAGE2)
    (COUPLE ?STAGE2 ?STAGE1))
  <(CKT74)>)
```

describes a task (COUPLER PLAN71) which couples two circuits and calls the result (CKT74). The action slot must be a function which gives an action given the input data. (Angle brackets surround tuples; "?" indicates a variable.)

I use the neutral word "task" to describe a problem to be worked on, because the concept is intended to be broad. Most problem solvers restrict themselves to actions which cause a (real or simulated) change in the state of the problem model. For example, in the blocks world, actions cause the positions of blocks to change. There are many actions which do not satisfy this definition, for example, "Think of someone who would be willing to lend you \$100"; and "While moving block A, avoid disturbing the blocks on top of it." The first example differs from the paradigm in that it causes no change in the world, but only the retrieval of some information. The second is an example of an action ("avoid...") which is executed solely as an influence on the execution of another action ("move..."). Another such "parasitic" action is "Wait here five minutes."

These classifications are indicated by using the predicates

(INFERENTIAL task-name action)

to indicate an information-retrieval task, and

(POLICY task-name action)

to indicate a task which is parasitic.

Task Scheduling--

Tasks are not executed as soon as they are created (unlike the right-hand sides of productions (Newell, 1973)). They may be postponed by rules using the "scheduling vocabulary." The basic predicate here is

(SUCCESSOR task-1 task-2)

which means (roughly) "task-1 must be executed or reduced before task-2." The scheduler examines SUCCESSOR formulas, then sets the state of a task using the predicate-calculus term

(TASK-STATUS task)

which is successively asserted EQUAL (in the data base) to PENDING, ENABLED, ACTIVE, and FINISHED. (The actual scheme is more complex. See McDermott (1976) for details.)

Task Reduction--

Even when a task is ENABLED, it cannot usually be executed immediately. If its action is not a primitive (i.e., does not have a LISP function to carry it out), it must be reduced to subtasks by the task reducer. Such tasks are called "problematic." The reducer first calls the theorem prover with a request of the form

(TO-DO task action output-data ?PLAN)

to retrieve a plan. A plan is either a single action, or a plan schema ("macro action") which expands into a set of subtasks. (For the details, see McDermott, 1976.) Either way, new tasks are created by the task reducer, and, for each one, a proposition of the form

(SUBTASK new-task problematic-task)

is recorded.

At any instant, the set of created tasks form a network defined by SUCCESSOR and SUBTASK formulas.

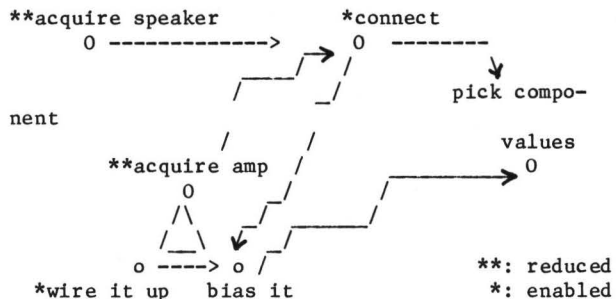


Figure 4 -- Task Network Midway through Designing a Hi-Fi

Here the arrows indicate SUCCESSOR formulas. A triangle pointing from low nodes to high indicates SUBTASK formulas; i.e., the low nodes are the subtasks of the high one.

Primitive Actions--

There are only a few primitive actions in the system. When the user is specifying a new domain, he uses the predicate

(MOD-MANIP task-name action delete-list add-list)

to define new "primitive" actions. (Cf. Fikes and Nilsson, 1971) For example, to define the effect of PUTON in the blocks world, one would write

```
(IMPLIES (ON ?X ?Z)
  (MOD-MANIP ?ANY-TASK (PUTON ?X ?Y)
    <(ON ?X ?Z)>
    <(ON ?X ?Y)>))
```

The built-in primitives include
> INFER -- for inferring a new fact

> FIND -- primitive information retriever

> NO-OP -- used for relabeling calculated data

> MONITOR -- a primitive policy (used for implementing things like prerequisite protection (Sussman, 1975)) to create an "interrupt" task when its monitored datum is erased.

Choice and Rephrasing--

Finally, there are the vocabularies belonging to the choice and rephrasing modules. The rephraser does not have a very rich vocabulary, a lack which reflects the shallowness of my theories regarding it. Rephrasing currently consists of creating a task with action

(REPHRASE recalcitrant-task action).

This falls in the category of "telling yourself what you are doing." If the user's domain-specific information does not include a plan for

rephrasing tasks of this sort, the system will produce a task with action (REPHRASE ... (REPHRASE ...)), for which there is a built-in plan to stop and ask for help.

The choice vocabulary is slightly richer. Rather than merely set up a "choice maker" task, NASL enters a special choice protocol. In this special module, the options (the set of competing plans) are set up by being mentioned in formulas of the form

(OPTION choice-name option-name
option-description)

Then a series of requests is made to the information retriever. Each is of one of these forms:

(RULE-OUT option-name)
(RULE-IN option-name)
(RULE-TOGETHER < -options- > new-option)

Retrieved formulas instruct the choice protocol to eliminate options, favor options, or compose options (in domain-specific ways). The cycle continues until only one option remains, or all options have been ruled out, or no progress has been made on the last loop. For example, the circuit-design knowledge has rules which suggest how to decide between alternative amplifier circuits, and when to try cascading them.

II.B Inferential Meanings of Vocabularies

The preceding survey of module vocabularies may be thought of as an informal description of the pragmatic component of their meanings -- how they are used by the modules. The richness of the system derives from the "inferential" meanings of the same symbols, determined by their interactions on the other side of the gatekeeper, that is, by the interactions of the rules containing them. These interactions are the medium of communication between modules.

An act of communication occurs when one module adds an expression to the data base from which an expression of interest to another module may be inferred. For example, one rule used by the designer is

(IMPLIES
(POLICY ?P
(CONSTRAIN ?DEV VOLTAGE-GAIN HIGH))
(TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
(MAKE OP-AMP)))

which defines the effect of the parasitic action CONSTRAIN as influencing the execution of MAKE. It suggests MAKE'ing an op-amp as a way of "solving" (MAKE AMPLIFIER) in the case where the amplifier's voltage gain is constrained to be high.

(Before going on, I should point out that not all rules relate two or more module vocabularies. Most TO-DO rules are "plan context" in-

dependent. Since in this paper I am emphasizing problem-solver state description, I will ignore the problem of domain-dependent vocabularies and their interactions.)

The most common classes of interactions which I have observed are these:

(1) TASK --> SUCCESSOR: Tasks which can interfere with each other must trigger rules to schedule them properly. For example, in electronics, there are rules to the effect that

"Any component-value selection subtask of a design task must follow every topology-altering subtask."

(The full rule, called SELECT-POSTPONE, may be found in McDermott, 1976). Many of the "critics" of Sacerdoti's (1975) NOAH may be thought of as built-in rules of this kind.

(2) POLICY --> TO-DO: Policies are defined as "parasitic" actions which influence other actions. One way this happens is by deduction of TO-DO formulas. I gave an example of this above.

(3) POLICY --> CHOICE rule: Another common way to define policies' effects is in terms of their influence on the operation of the choice protocol used to pick among plans. For example, one amplifier-design rule says:

"In choosing between a one-stage common-emitter and a multi-stage, if the bandwidth is CONSTRAINED to be HIGH, RULE-OUT the one-stage."

(Cf. DIFF-CE-N-STAGE, in McDermott, 1976)

(4) TASK --> SUBTASK: Sometimes task reduction occurs entirely via inferential rules; the task reducer just ignores an already-reduced task. For example, bias and coupling plans for multi-stage circuits overlap in their duties. A typical rule from a coupling plan says,

"The tasks for coupling to the second stage also do the work of biasing the base of the second-stage transistor."

(By the way, to make these rules effective, there is another rule, of "interaction type (1)," which says, "Do coupling before biasing." (Cf. Figure 4.) See the rules COUPLE-BEFORE-BIAS and CE-DIR-VOL-COUPLE-PLAN in McDermott, 1976.)

There is nothing special about this list of common interactions; any rule may be used that can be handled by the inferential system. (Of course, we can't expect genius-level insight from the data-base machinery. For instance, we must help it out by telling it whether to use a statement of the form (IMPLIES p q) in a forward or backward direction. Cf. Hewitt, 1972.)

III Results

The NASL system has been applied to two rather different tasks: electronic-circuit design and the blocks world. Both applications are still being debugged. The first domain requires many rules implementing theories of design and electronics; the system currently possesses about 350 rules defining design, elementary electronics, typical circuits, standard biasing plans, and much more; even so, its knowledge is pitifully sketchy compared to what a technician knows. The blocks world was studied for a different reason: as a way of comparing NASL with Sacerdoti's (1975) NOAH; in this domain, the knowledge requires about six pages of rules (on the order of 30 rules).

NASL plus electronics and design rules is called DESI. DESI has never designed a circuit all the way through. It has, however, given ample opportunity for testing the power of NASL's vocabularies. The system at this writing is capable of doing most of the steps in some simple design tasks. For example, its theory of design specifies a standard "design rephrasing plan" which transforms design problems in ways varying from splitting conjunctive problems into their conjuncts, to translating signal-conversion problems from the time to the frequency domain. (McDermott, 1976, 1978)

The theory of design is fairly domain-specific. That is, there is no general theory of putting to use a new kind of element; instead, there are many prepackaged partial plans for various tasks, and the main job of the designer is to coordinate them. It would be nice if some of the inferential interaction rules of the last section were deducible from knowledge about new devices, but accomplishing this seems very hard.

After NASL was devised for this task, I tried applying it to the rather different blocks world, to see how general it was. My model was the NOAH program, which has several built-in constructs I thought could be expressed as rules. Some could; some couldn't.

For example, the notion of "prerequisite" (Sacerdoti, 1975; Sussman, 1975) is not built into NASL. It can be defined by rules which say

"IF T1's effect P is a prereq of T2, then T2 is a SUCCESSOR of T1, and there is a policy TP to protect P until T2 is begun."

(Protection must be further defined in terms of MONITOR.) The notion of prerequisite can then be used in two ways: passively, to catch "protection violations" (Sussman, 1975); and actively, as a trigger of critics like "Resolve Conflicts." (Sacerdoti, 1975)

One irreducible difference between NASL and NOAH is NASL's inability to reduce a task before its predecessors are reduced. To be able to do this, one must have at least a partial model of

the problem state after the predecessors are finished. NOAH can assume such a model is well defined, because NOAH assumes actions are defined as state changes. NASL is pessimistic about the existence of such a model; it never reduces a task until it is time to start executing it.

It is instructive to think of this as a deficiency in vocabulary. That is, there is no action of the form "reduce a task" which NASL can carry out. There is also no built-in term (BEFORE task) which would designate the problem model in which to carry out the reduction (but I believe this could be defined now in terms of things like MOD-MANIP without extensions to NASL). NOAH, on the other hand, has a more limited notion of "task," in which all tasks are non-inferential and non-parasitic, have foreseeable effects, and do not compute results. This enabled its author to separate planning from execution cleanly.

IV Conclusions

I have presented a model of using a representational system, especially its use by a problem solver to represent its state. The model assigns to each module of a problem solver a vocabulary consisting of symbols with special meanings not derived from inferential interactions. I applied the model to the description of the NASL system (McDermott, 1976), and sketched a comparison with NOAH. (Sacerdoti, 1975)

This model has the following advantages:

(1) Expository clarity -- In describing a problem solver, one can factor the description into a formal, representational component, and a "pragmatic" representation user.

(2) Modular implementation -- The two pieces can be implemented and optimized separately. (McDermott, 1976)

(3) Ease of comparison -- Many differences between programs can be expressed as a difference in vocabulary.

(4) Ease of experimentation -- To add a new module is to add a new vocabulary, as far as the representation system is concerned. There are no unnecessary restrictions on form, and no need for a set of special communication channels between a new module and the old ones.

The model brings out these points about NASL

--

(1) It has an unconstrained notion of problem, the "task." Tasks may be "inferential" or "parasitic," and may compute and receive data from other tasks.

(2) NASL interleaves planning and execution tightly. Planning amounts to the execution of a problematic task.

(3) NASL is incapable of "lookahead" task reduction. There is no built-in term for the state of affairs after an action.

(4) More generally, NASL relies on "shallow reasoning" for thinking about plans. It does not contain a complete axiomatization of any programming language, and cannot prove the correctness of any of its plans. The reason it is successful in spite of these limitations is that it represents its current plan in a very redundant way; almost all features of interest can be retrieved quickly. This means that interesting advice about the domain is likely to be efficiently representable. In a complex and uncertain world, this is probably the best we can do.

The model is still under development. One area in which it needs work is in a general characterization of modules which would allow us to be as unrestrictive yet precise in describing the "pragmatic" component of symbol meaning as predicate calculus allows us to be in describing the "inferential" component. So far, we must make do with English descriptions like those of Section II.

One possibility is to use a production system for this. That is, a problem solver would be described by a set of "state transition" rules whose left-hand sides described problem-solver states and whose right-hand sides specified changes in terms neutral enough to cover all problem solvers. Modules would correspond to groups of productions sharing a vocabulary. A problem solver would then consist of two complementary pattern-directed systems, one to do inference, the other to do action. This has a certain pleasing symmetry.

References

1. Davis, Randall. Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases. Memo AIM-283, Stanford University Artificial Intelligence Laboratory, 1976.
2. Fikes, R.E. and Nilsson, Nils J. STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 1971, p. 189.
3. Hewitt, Carl. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Technical Report 258, MIT Artificial Intelligence Laboratory, 1972.
4. McDermott, Drew. Flexibility and efficiency in a computer program for designing circuits. Ph.D. thesis, MIT Artificial Intelligence Labo-

ratory, 1976.

5. McDermott, Drew. Deduction in the pejorative sense. Forthcoming, 1977.

6. McDermott, Drew. Circuit design as problem solving. Submitted to IFIPS Working Conference on AI and Pattern Recognition in Computer-Aided Design, 1978.

7. Moore, Robert C. Reasoning from incomplete knowledge in a procedural deductive system. Technical Report 347, MIT Artificial Intelligence Laboratory, 1975.

8. Newell, Allen. Production systems: models of control structures. In Chase, W.C. (Ed.) Visual Information Processing, Academic Press, New York, 1973, p. 463.

9. Sacerdoti, Earl D. A structure for plans and behavior. Technical Note 109, SRI Artificial Intelligence Center, 1975.

10. Shortliffe, Edward H. Computer-Based Medical Consultations: MYCIN, American Elsevier Publishing Company, Inc., New York, 1976.

11. Sussman, Gerald J. A Computer Model of Skill Acquisition, American Elsevier Publishing Company, Inc., New York, 1975.