Prolog for applications programming

by W. G. Wilson

This paper discusses the problems and benefits of using the Prolog language to write application programs. Much attention is currently focused on expert-systems shells, which play a role in artificial intelligence (AI) systems similar to that of application generators in more conventional applications. The Prolog programming language embodies many of the features found in these shells, while providing a relatively general and complete programming language. MVS performance tuning is used as an application that typifies a broad class of applications suitable for implementation in this language. Some of the difficulties that had to be overcome to use the language are presented, with their solutions.

Prolog is a computer language that has aspects that make it different from such other languages as PL/I, Pascal, COBOL, or FORTRAN. This difference goes deeper than syntax or a set of primitives. It is at the very core of the language, in what a Prolog program means. A Prolog program can be viewed as a set of logical axioms, where executing a Prolog establishes a proof that a certain desired conclusion follows from that set of axioms.

The name Prolog stands for Programming in Logic. The language was invented in the early 1970s as a practical implementation of a theorem prover for a subset of first-order logic.[1-3] Intended as an implementation language for artificial intelligence (AI) applications, Prolog incorporates many of the features of other languages, such as LISP, and deals primarily with tree structures of which the LISP list is a subset. Prolog programs are usually recursive.[4]

After an introduction to the Prolog language, a particular application is described that is typical of a broad class of applications suitable for implementation in this language. Some of the difficulties that had to be overcome in order to use the language—and their solution—are presented. In this paper, the term *author* refers to the analyst and application programmer, and *user* refers to the user of an application. When the application is an expert system, other writers sometimes use the terms *knowledge engineer* and *client* in place of author and user. The term *logic* refers to the relationship that holds among data structures. This usage follows from the fact that Prolog is based in mathematical logic. It is not used synonymously with *control*, as is customary when talking about procedural languages.

Prolog

The Prolog language has been shown to be useful for such complex problems as analysis of hardware and software.[5,6] It also has useful software engineering aspects for less complex problems.[7] Several people have noted the usefulness of Prolog for implementing expert systems.[8-12] The language provides a basis for utilizing natural parallelism,[13-15] and variations of the language apply to non-AI-type applications.[16,17] Prolog has had many different implementations with different syntax and built-in procedures, but they all involve the same essential concepts that make Prolog recognizable in many forms. Prolog is probably the best known of several languages that utilize theorem proving in logic as a programming tool in the general area called *logic programming*.[18-20]

Procedural versus declarative meaning. One might question whether there truly is a relationship between theorem proving and programming. In fact, a close relation exists because Prolog knits together a *procedural* (or programming) and a *declarative* (or logical) view of a program, from which process comes its value.[21] This relationship can be illustrated by contrasting the meanings of procedural programs with the meanings of logic programs. The meaning of a Pascal program lies in the sequence of steps that the computer performs, given certain input. The relationship of input to output and the procedural flow are mingled together and cannot be separated. A demonstration that the program actually realizes its English language specification can be difficult, however.

A Prolog program, on the other hand, has two clear and distinct meanings. One meaning is the procedural one which, like Pascal, instructs the computer on the sequence of steps it is to take. The declarative meaning has nothing to do with a computer. A Prolog program can be read as a set of statements (called *axioms* or *clauses*) about the real world that are either true or false. Executing a Prolog program, from this viewpoint, amounts to showing that a certain statement can be deduced from the logical statements that make up the program. A *proof* of the goal statement is constructed and demonstrates that the goal logically follows from the axioms that are the Prolog program. This is the fundamental essence of Prolog.

Prolog and theorem proving

It is not necessary to understand theorem proving to use the Prolog language. However, a little discussion of the origins of the language may help put it in perspective with other languages and with expert-systems shells.

At the time Prolog was introduced, there were complete theorem provers for the first-order predicate calculus, which is the logical system developed by logicians to express the bulk of deductive reasoning, a central problem in AI research.[22,23] Although a theorem prover can in principle deduce any valid proof, the time required may be excessive. At each step in a deduction there may be many ways to proceed. Since any of the possible deductive paths is logically valid, the decision about which path to take does not affect the correctness of the result. Rather, it is a question of the efficiency with which a correct result is obtained. Deciding which way to proceed in the deduction is done by the control component of the theorem-proving system.

On this point there are two main divergent opinions. One is that a general-purpose control strategy or set of strategies can be devised. The other is that to be efficient, the author must be able to specify control information that relates to the specific problem the program is to solve.

The declarative meaning of a Prolog program makes it different from procedural programs.

Prolog gives this second facility to the author. The general framework of a Prolog proof is that of backward reasoning from a desired conclusion to unconditional facts. Within this framework, the author has explicit control over the order in which axioms and subgoals are tried. This gives the author of a Prolog program as much control over how the program executes as a Pascal programmer has. Because they are based in logic, however, Prolog programs also have a declarative or logical meaning, which Pascal programs do not have. The author can separate the issue of correctness from the issue of efficiency. This makes the author's tasks in each of these areas simpler than when they are mingled together, as they are in procedural languages.

The existence of the declarative meaning of a Prolog program is what makes it different from strictly procedural programs. The problems and advantages described here arise from this feature of the Prolog language. Because Prolog has both logic and programming aspects, certain nonlogical features were introduced into the language to take care of such pragmatic concerns as reading and writing data. To reap the maximum benefit of the logic-programming paradigm, an author must strive to isolate these nonlogical aspects of a program.

From this declarative viewpoint, executing the Prolog program establishes a logical proof that a goal follows from the axioms. Given the axioms that

make up the Prolog program, the result is true regardless of the method used to establish the proof. The method is called the *proof procedure* or *strategy.*

Prolog uses one particular strategy, but any other sound strategy could be used on the same program to reach a true conclusion. This is like not having to consider the order in which Pascal statements are executed, but still being guaranteed a true result, because a proof is a proof no matter how it was established. The Prolog strategy is termed *top-down* or *backward chaining*, because the direction of reasoning is from the desired conclusion to unconditional axioms.

Programming with Prolog

Writing a Prolog program. Consider first the following paragraph of ordinary English prose:

New York City gets its water from a variety of sources. One of these sources is the Ashokan Reservoir in upstate New York. This reservoir occasionally dries up, at which times it is empty. Whenever a city gets its water from an empty reservoir, that city should limit water use.

This paragraph expresses a variety of thoughts, so that a person after reading it is in a position to answer such questions as the following:

- Should New York City ration water use?
- Where does New York City get its water?

The simplified logical content of this paragraph could be expressed in the following three sentences.

1. New York City gets its water from the Ashokan Reservoir.
2. The Ashokan Reservoir is empty.
3. A city should ration water use if it gets its water from some reservoir and that reservoir is empty.

Notice that the first two sentences express simple facts, whereas the last sentence expresses a conditional truth. When expressed in Prolog, each sentence is called a *clause.* The conditions following the word "if" are called the *body* of the clause. The first two sentences have no body, in which case the body is said to be "empty."

There are many ways of formalizing these sentences in logic. The degree of complexity of the formalization depends on the questions that are expected to be answered. At the least detailed level, for example, the first sentence could be considered as simply a statement with neither generality nor structure. It could be used to answer only the question "Does New York City get its water from the Ashokan Reservoir?"

At the most detailed level, the sentence could be considered to express a variety of things. There are two objects, a reservoir (the Ashokan) and a city (New York City). There is another object (water) and a relation (gets) that says one thing (New York City) is getting another thing (water) from the third thing (the Ashokan).

The usual case is somewhere between least and most detailed levels. In this example, consider two objects, the Ashokan Reservoir and New York City, and make "getting water from" the relation between these two objects. This relation can then be used to express other facts, such as where Albuquerque gets its water (i.e., the Rio Grande and underground aquifers).

The second sentence ascribes the property of being empty to the Ashokan Reservoir. The third sentence gives a rule for when a city needs to ration water use. Definite but unspecified objects (such as "some city") play the role of variables.

These sentences about rationing water use can be represented in Pascal data structures and programs, but the Prolog language allows particularly simple representations of these sentences. The normal form of a Prolog relation is called a *predication* and is expressed as follows:

<relation name> (<list of objects being related>)

As in other languages, variables must be distinguished from constants and relation names. In these examples, the variables begin with uppercase letters. The words "if" and "and" are represented by <- and &. Letting A, B, and C be arbitrary predications, the general form of a Prolog clause is as follows:

A <- B & C

which is read "A is true if B and C are true." As mentioned previously, the conditions (B & C) are called the body of the clause. The conclusion (A) is termed the *head.* There may be any number of conditions in the body.

The following Prolog clauses correspond to the previously given sentences:

1. empty(reservoir(ashokan)).
2. city_gets_water_from_reservoir(new_york_city, ashokan).
3. should_ration_water_use(Some_city) <- city_gets_water_from_reservoir(Some_city, Some_reservoir) & empty(reservoir(Some_reservoir)).

In addition to this normal form, most Prolog implementations allow a *syntactic sugaring*. That is, by suitably instructing the Prolog parser, the following more English-like syntax can be used:

1. the ashokan reservoir is empty.
2. new_york_city gets its water from the ashokan reservoir.
3. Some_city should ration water use if Some_city gets its water from Some reservoir and Some reservoir is empty.

Difference between English and Prolog. The English-like nature of this example is deceptive. The declarative meaning of this program is straightforward, as can be seen by reading each statement as the corresponding sentence. However, Prolog does not understand the English, so that editorial variations or equivalent expressions do not work. "The Ashokan Reservoir is out of water" cannot be used in place of "The Ashokan Reservoir is empty." A person can answer questions about the paragraph because a human reader understands what is being said. Prolog performs only formal operations on the axioms it is given. Within these limitations, however, Prolog operations are such that the result is true (as determined by a person) if the axioms are true (as determined by a person).

The particular symbols used have no meaning to Prolog. One could replace "should_ration_water_use" with "apples" and "gets_its_water_from" with "oranges," and it would make no difference to Prolog, which would still carry out its operations mechanically. In Prolog as in other languages, whether the program and the results have any meaning depends strictly on the observer. One would have to know that "apples" really means that a city should ration water use, and that "oranges" really means that a city gets its water from some particular place, before the program or its results would have meaning.

Of course, this limitation affects all programs, not just Prolog programs. The meaning of symbols depends on the human observer and not on the computer. Prolog cannot make associated judgments about the content or meaning of the symbols used in a program, and can deduce nothing other than what is strictly given. It has none of the general or commonsense knowledge that a person has.

Prolog uses a pattern-matching process called unification to give values to variables.

Executing a Prolog program. We have seen how a Prolog program looks and how it is read. We now discuss how the program executes. Each relation in a Prolog program is a *procedure*. In the water-rationing example, there are three procedures: "empty," "city_gets_water_from_reservoir," and "should_ration_water_use." In this example, there is only one clause for each procedure, but there could be more. One might also know that Kingston gets water from the Catskill Reservoir:

city_gets_water_from_reservoir(kingston, catskill).

Then there would be two clauses for the procedure "city_gets_water_from_reservoir." Considering the whole world, there could be thousands. In Prolog, a procedure call is also known as a *goal* to be proved. The body of a clause is a sequence of procedure calls. It is also called a goal. The body of the only clause in the procedure for "should_ration_water_use" is a sequence of two procedure calls, "city_gets_water_from_reservoir" and "empty." Prolog procedures contain only calls to other procedures.

A Prolog procedure call may either succeed or fail. If it succeeds, a proof has been established for a true instance of the goal; if it fails, no proof has been established.

Prolog does not have an assignment statement. Instead it uses a pattern-matching process called *unification* to give values to variables. When a procedure is called, each clause in the procedure is tried in turn. The variables and data structures in the goal are matched against those in the head of the clause. A

variable matches anything, and a constant matches only the same constant.

For the following example, the goal

city_gets_its_water_from_reservoir(R, ashokan)

matches

city_gets_its_water_from_reservoir(
new_york_city, ashokan)

In this process of matching, the variable R is assigned the value "new_york_city."

When a clause is found whose head matches the goal, the process is repeated on each of the goals in the body. If they all succeed, the goal that matches the head of this clause also succeeds. All of the matches that occur usually cause some of the calling variables to become bound by unification. This is the way results are generated.

Suppose the goal is the following:

city_gets_water_from_reservoir(new_york_city, R).

There are the following two clauses in this procedure:

city_gets_water_from_reservoir(new_york_city, ashokan).

city_gets_water_from_reservoir(kingston, catskill).

Trying the first clause, the constant "new_york_city" matches itself in both the goal and the head. The variable R in the goal then matches "ashokan." Because this clause has no conditions to try, the goal immediately succeeds.

Suppose the goal is

city_gets_water_from_reservoir(kingston, R).

The match of "new_york_city" with "kingston" fails, which causes the second clause to be tried. This clause succeeds, matching the goal variable R with catskill.

Where a goal consists of a sequence of procedure calls, one subgoal may succeed while a subsequent subgoal fails. In this case, *backtracking* occurs, whereby an alternative solution to the previously succeeding subgoal is sought. Although we do not discuss the details of backtracking, it is sufficient to understand that, in order for a goal to succeed, all procedure calls in the body of a matching clause must succeed.

If the goal is

should_ration_water_use(Some_city)

the corresponding rule is invoked. So far, "Some_city" does not have a value. The rule has two subgoals in its body. Each of these must be satisfied in turn as follows:

The programmer does not have to be concerned about the control of the program to establish correctness of the results.

city_gets_water_from_reservoir(Some_city, Some_reservoir) &
empty(reservoir(Some_reservoir)).

First, the procedure "city_gets_water_from_reservoir" is called. Then the procedure empty is called.

Suppose the goal

city_gets_water_from_reservoir(Some_city, Some_reservoir)

first matches the clause

city_gets_water_from_reservoir(kingston, catskill)

Now the following variables have values:

- Some_city = kingston
- Some_reservoir = catskill

The next goal is

empty(reservoir(catskill))

Given what is in the program, this cannot be proved. Therefore, the call fails, which causes backtracking. The previous goal has the following solution that matches the clause:

city_gets_water_from_reservoir(new_york_city, ashokan)

Therefore, the variables get the following new values:

- Some_city = new_york_city
- Some_reservoir = ashokan

This time the subgoal "empty(reservoir(ashokan))" succeeds. We now have the following, because both goals have succeeded:

```
should_ration_water_use(new_york_city) because
    city_gets_water_from_reservoir(new_york_city,
    ashokan) &
    empty(reservoir(ashokan)).
```

Prolog also allows tree structures as well as variables and constants, which makes the matching process quite powerful. However, we do not discuss tree structures in this paper.

Separate logic from control. We have shown a way of reading a Prolog program as a logical statement and another way of reading it as a series of procedure calls. It is this logical meaning that makes Prolog different to work with. The author of a Prolog program can switch between these two ways of viewing a program, which makes some activities much easier. For example, from the declarative viewpoint, the correctness of a Prolog program is independent of the particular method used to carry out the proof. That means that the programmer does not have to be concerned about the control of the program to establish correctness of the results. The programmer does, however, have to pay attention to the control aspects to improve *efficiency*.

The previous example shows that the declarative meaning of the program is correct when it answers (in effect): Yes, New York City should ration water use. This is true without regard for either the order in which the subprocedures are called or the order in which the sentences appear in the program. Of course, one may disagree with the assumptions, in which case one may also disagree with the conclusion. However, if one agrees that the assumptions are true, then one must agree that the conclusion is also true.

This program can answer questions about empty reservoirs (Which reservoirs are empty?) and about water supplies of cities (Which city gets its water from which reservoirs?). The program is not limited to the one question about New York City's limiting its water use. Logic programs are more general in this sense than procedural programs. They are also very modular, for the same reason, in the sense that everything is written as many small procedures.

These procedures can also be used in the relational data base style. The one procedure "gets_its_water_from_the_reservoir" can tell either which reservoir New York City gets its water from, or which cities get their water from the Ashokan Reservoir, or check whether New York City gets its water from the Ashokan Reservoir, and so on. Which variables are input and which are output can vary from one call to the next. This property of Prolog programs, referred to as *relational reversibility*, is often useful in reducing the amount of code that has to be written.

The application selected is that of assisting in the performance tuning of MVS.

In Pascal, for example, separate functions have to be written for finding reservoirs of cities and for finding cities that use reservoirs, whereas in Prolog there is but one relational procedure.

A performance-tuning application

The application selected for exploring the potential of Prolog programming technology is that of assisting in the performance tuning of MVS, IBM's primary operating system for large systems. For systems as complex as MVS on large mainframes, there are many ways to approach the problem of performance tuning. Therefore, our approach is to use methods and knowledge provided by an expert in the field of performance tuning, D. W. Hunter. Thus, the application may be said to be an expert system.

This application is restricted to TSO environments, where secondary-storage (DASD), control-unit, and paging problems can cause degradation in response time. The recommended solutions are those that take relatively long times to implement, such as balancing dataset activity or installing more main storage. The system does not perform minute-by-minute adjustments of operating system parameters such as target multiprogramming level or number of active batch job initiators.

The data on which the recommendations are based are the Resource Management Facility (RMF) trace information,[24] the System Generation (SYSGEN) configuration information,[25] and information known only to the systems programmer, such as which performance groups identify TSO users.

The RMF data are produced on MVS in variable blocked spanned records. This file organization is not supported by the VM/CMS operating system on which the tuning system runs. Therefore, a PL/I program was written to convert the file format to one supported by CMS. The PL/I program also normalizes all data units. That portion of the system written in PL/I, though a small part functionally, required nearly one third of the total development time. The programmers involved were equally skilled in both Prolog and PL/I.

The system is organized into three levels corresponding to the expected performance experience of its user.

Because SYSGEN information is maintained on line in a file format already supported by CMS, it did not require format translation, but was read directly by Prolog programs.

Two major versions of the expert MVS tuning system were produced. The first version dealt with MVS/370 only. The second version was expanded to handle also MVS/XA. This was a significant conversion because there were major input data format and content alterations, as well as changes to the tuning methodology embodied in the program. Prolog proved remarkably valuable in minimizing the conversion time.

The features of Prolog that contributed to making conversion easier are *extendability* and *modularity*.

Extendability. The XA version of RMF and SYSGEN data had records not present in the System/370 version, and the meaning or content of some of the fields in other records was changed. New records could be handled by simply adding clauses that defined these new records and others that expressed the knowledge of how they were to be used. Existing knowledge about information valid in System/370 systems did not require changing. The knowledge was cumulative. For records whose meanings had changed, definitions were given with conditions added specifying the level to which they applied.

Modularity. Because everything in Prolog is done in essence by a subroutine, replacing or extending the logic was simplified. In contrast, the PL/I procedures we had to modify had much in-line code that had to be inspected and modified whenever the control or data references were inappropriate.

Capabilities. The system is organized into three levels corresponding to the expected performance experience of its user.

- *Generalized level.* The most elementary level, this is a simple step-by-step process for users who know nothing about performance tuning in general or about the problems specific to the MVS system for which advice is requested. At this level, a user can ask such questions as "Is there a TSO response time problem?" and "What should I do about it?". See Figure 1A. The user gets only one recommendation about the most troublesome problem, even though more than one recommendation may be applicable. Without knowledge of the tuning process, the user is not in a position to evaluate more than one possible solution. The next level of detail can be used to investigate other problems if desired.
- *Localized level.* This level is for users who know about the performance problems of a particular system and who want to direct the tuning activity in more detail, focusing it on one of eleven primary problem areas that is suspected of causing the current performance problem. At this level, more than one problem can be exposed if multiple performance problems exist in a particular system. The user can ask questions such as "Is there an overloaded DASD device or control unit?" and "Is the system demand paging rate too high?". See Figure 1B.
- *Data extract.* At this level systems programmers who are familiar with the performance tuning of large systems can use the tuning system in summary mode as an easy-to-use window on the basic performance data. They can pursue a problem

outside the current scope of the tuning system, using their own expertise. This level provides such information as the Start Subchannel (SSCH) rate for a particular device or for all devices on a control unit, and response time for specific MVS performance groups and periods. See Figure 1C for an example of the type of information available.

System queries. During the course of attempting to identify a performance problem and recommend a solution, the tuning system may need information that is not currently part of its knowledge base and that cannot be inferred from existing rules. One example is RMF trace data. The tuning system does not come prepackaged with tuning data about a particular system on a particular day. When it needs this information and it is not currently available in the active workspace, the user is asked where to get the information. Then the appropriate file is consulted.

There may be some information that only the user can provide. For example, each MVS installation gives separate *performance group* identifiers to different departments for various accounting and administrative uses. RMF performance information is collected on the basis of these *performance groups*. The tuning system needs to know which of the performance groups are to be considered in establishing response-time characteristics. This information is available only from the installation's systems programmer.

Figure 1 (A) Example of a menu for common use, (B) example of a menu for an experienced user, (C) example of a menu for use by a performance expert

```
                MVS/XA PERFORMANCE TUNING EXPERT SYSTEM
            SYSTEM A1 INTERVAL:M021  03-22-85  13:00   SCREEN: MAIN

        1. Start/restart a consultation.

        2. Is there a long TSO response time problem?

        3. Is there a solution to the long TSO response time problem?
           (This choice applies all available tuning knowledge to locate
           a problem.




           To examine the solution possibilities separately, press PF7 now.)

           ENTER YOUR CHOICE OR PRESS A PF KEY ====> _
        1-HLP 2-BRW 3-QUIT 4-EXIT 5-REV 6-NXT 7-ALL 8-PRO  12-CMS
```

```
                MVS/XA PERFORMANCE TUNING EXPERT SYSTEM
          SYSTEM A1 INTERVAL:M021  03-22-85  13:00   SCREEN: SLTN

 1. Show any overloaded device (of the top 10 candidates).
 2. Show any DASD chpid with heavy tape usage.
 3. Show any overloaded chpid causing DASD delays.
 4. Show if there are any long channel programs on a DASD chpid.
 5. Show any excessive reserve delay on a device.

 6. Show whether the working set is too small.
 7. Show if the system total paging is too much.
 8. Show whether the logical swaps are being physically swapped.

 9. Show whether there are any overloaded control units.
10. Show whether there are any chpids with high utilization.
11. Show whether there are any chpids with high service time.


    ENTER YOUR CHOICE OR PRESS A PF KEY ====> _
1-HLP 2-BRW 3-QUIT 4-EXIT 5-REV 6-NXT 7-ALL 8-PRO  12-CMS
```

```
                MVS/XA PERFORMANCE TUNING EXPERT SYSTEM
          SYSTEM A1 INTERVAL:M021  03-22-85  13:00   SCREEN: SSCH

1. Show the SSCH rate for this DASD.  ==> ____

2. Show the SSCH rate for this string of DASD.  ==> ___

3. Show the SSCH rate for this physical control unit.  ==> __

4. Show the SSCH rate for this logical control unit.  ==> __

5. Show the rate that SRM issued STCPS to sample chpid busy.




   ENTER YOUR CHOICE OR PRESS A PF KEY ====> _
1-HLP 2-BRW 3-QUIT 4-EXIT 5-REV 6-NXT 7-ALL 8-PRO  12-CMS
```

Figure 2 Example of a question the system asks the user

```
          MVS/XA PERFORMANCE TUNING EXPERT SYSTEM
SYSTEM A1 INTERVAL:M021  03-22-85  13:00   SCREEN: TSOPGS

What performance groups are for TSO in this interval?

 Performance group ===> ____
 Performance group ===> ____
 Performance group ===> ____




1-HLP 2-BRW 3-QUIT 4-EXIT 5-REV 6-NXT 7-ALL 8-PRO  12-CMS
```

In these cases, the tuning system asks the user to supply the missing information. This is done by means of a *query screen*, as may be seen in Figure 2.

System responses. The system locates a problem, gives a recommendation, and justifies its recommendation. The justification is a logical explanation of the problem and the recommendation in terms meaningful to the user. See Figure 3 for an example. The degree of detail and the terms used are under author control and can be made dynamically responsive to a particular user's interests or background. One user may want to know only which buttons to push to set things right, whereas another may wish to know the device activity characteristics of those strings of devices that are causing the problem. In this particular application no significant use was made of this dynamic capability. Answers are given at one level of detail. However, the capability is there in case of future need.

Capturing the performance knowledge. The tuning system does not contain an explicit performance model of MVS. It operates at a superficial level, by mimicking the observed processes of human experts. These experts undoubtedly do have a mental model of MVS to which they refer when questions go outside established procedures. In that sense the tuning system is incomplete. Although Prolog is suitable for implementing such a model, the design decision not to do so was originally made in order to be more faithful to the capabilities and mode of behavior of the broad class of applications we were trying to represent. Likewise, most standard data processing

Figure 3 Example of results the system shows the user

```
                MVS/XA PERFORMANCE TUNING EXPERT SYSTEM
          SYSTEM A1 INTERVAL:M021  03-22-85  13:00   SCREEN: RSLT

(PRESS PF2 TO BROWSE COMPLETE RESULTS FILE)
 There exists a DASD solution to reduce response time.
 An overloaded device exists.
 Use GTF analysis to spread workload for device 0570.

    Device 0570 is a 'problem device'.

    Device 0570 is overloaded during interval 13:00.

       The SSCH rate to device 0570 is 5.09/second.

       A SSCH count greater than 5/second is too high.

       The average queue length for device 0570 is 7.277.

    An average queue length (enqueue count) greater than 5 is too high.
1-HLP 2-BRW 3-QUIT 4-EXIT 5-REV 6-NXT 7-ALL 8-PRO  12-CMS
```

applications as well as expert systems do not incorporate a detailed model of an enterprise when producing a loss/earnings report, for example, or when diagnosing a system failure.

However, experience with the existing tuning capability has convinced us that such a deep model of an MVS system should be incorporated. It would be useful for predicting the effects of suggested changes to the installation being tuned and in providing more tutorial types of explanations. It is important that the language used to implement the application be able to support both types of knowledge: superficial behavior copied from an expert and a deep model of the domain. Without such a model, the system cannot be considered to understand a problem in any significant sense.

Prolog is a language whose basic concepts are powerful enough to provide the underlying representation for a variety of knowledge-representation methods. In our case, a simplified version of the language was used to represent directly the rules provided by the tuning expert. As in most applications, programmers and analysts are used as intermediaries between the expert and the machine. A fair degree of expertise in putting together applications, including what the screens should look like and how to make procedures more efficient, is required with applications of the type we were emulating. After studying various application generators and expert-systems shells, we are convinced that the role of programmer/analyst still exists. New tools may shorten the path from expert to program, but it still requires a programmer/analyst or *knowledge engineer* as intermediary.

Application development

Production of rules. The production of the rules for the MVS tuning application proceeded in three stages:

1. We interviewed the expert on particular tuning issues and took detailed notes of the interview.
2. We then organized the notes and expressed the expert's thoughts in complete sentences. We reviewed the resultant transcript with the expert for accuracy and completeness. This occasionally resulted in revisions of the expert's original statements.
3. The sentences were translated directly into Prolog and validated on sample data by comparing program results with the expert's results.

This process was repeated several times to add more scope. A general review was performed in the conversion from MVS/370 to MVS/XA. Here we used the extendable nature of Prolog programming advantageously.

Programming environment. During this process, common functions were extracted from the Prolog procedures and incorporated into a set of utilities that made programming significantly easier. These constituted a program development environment. Although developed independently in response to perceived need for the task at hand, this programming environment for Prolog is similar in many respects to facilities available in Augmented Prolog for Expert Systems (APES)[26-28] and in the Expert System/Development Environment.[29]

Metarules. We are able to keep the rules simple and logical by providing a metarule capability. Metarules look exactly like the Prolog clauses that express the performance-tuning knowledge, but they express control and user interface information. Essentially, they say how the actual performance rules are to be used, what intermediate results are to be remembered for future use, when questions are to be asked of the user, and what information is to be given to the user for an explanation.

For example, suppose that the rules about water use do not include the information about which reservoirs are empty. It may be that this information is available only from the user of the system. Then the metarule can be added to tell the system to ask the user whether it needs this information: askable (empty(reservoir(*))).

When the following clause is invoked,

```
should_ration_water_use(Some_city) <-
  city_gets_water_from_reservoir(
  Some_city,Some_reservoir)&
  empty(reservoir(Some_reservoir)).
```

the subgoal empty(reservoir(ashokan)) cannot be proved from existing information. Because of the metarule that says this subgoal is askable, the user will be asked whether it is true. An affirmative reply causes the subgoal to succeed. This in turn causes the initial goal regarding rationing water use to succeed also. This is not a direct feature of Prolog, but is provided by our program development environment. The author can specify a full-screen interface to be used to ask the question or give a procedure that can generate a line-oriented message. However, the details of this feature are not given here.

In addition to asking the user for information, metarules are available to remember intermediate results, to control processing, and to generate the form of a result. The interesting thing about using metarules is that the knowledge remains expressed as pure logic, and the metarules themselves are strictly declarative. Yet such pragmatic considerations as user interface and control are handled satisfactorily. Otherwise these usually require imperative or nonlogical constructs.

Translating from English to Prolog. With an eye to translating into Prolog, the English sentences are either simple statements or compound conditionals of the form IF . . . AND . . . ARE TRUE, THEN . . . AND . . . ARE TRUE. In the following example, xx and yy are specific values not reportable here.

Example: A device in the list of problem devices is overloaded if the start-subchannel rate for the device is greater than xx and the average enqueue delay is greater than yy.

After the interview with the expert had been put in complete English sentences and verified by the expert, the act of coding the rules in the Prolog language began. Because of the strong correspondence between Prolog forms and the English sentences, this was a straightforward process that proceeded as follows:

- A list of all the types of objects mentioned by the expert was drawn up.

Example: device list of problem devices, control unit, page swap, string of devices, etc.

- All relevant properties of these object types were listed, as determined by the expert interviews.

  Example: device start-subchannel rate, device response time, device average enqueue delay, etc.

- All mentioned relationships among these objects were identified.

  Example: systems can share control units.

- The representation of an object of each type was determined. For example, because a device is referred to in MVS/370 by a three-character address, we represent devices by their MVS/370 addresses. An address was realized as a record having the Channel (CH), Control Unit (CU), and specific device (D) as three subfields. The Prolog data structure representing this record is CH.CU.D. Prolog pattern matching can then be used, for example, to refer to any device on channel 1 by using the term (1.*.*).

- Formal names were given to properties and relations. These were selected so as to be meaningful to subsequent authors or others having cause to read the program.

  Example: rmf_device_specific_start_subchannel_rate, overloaded_device, . . .

- The English sentences were rephrased, using their formal counterparts:

```
overloaded_device IF
  rmf_device_specific_start_subchannel_rate >
  XX AND
  rmf_device_specific_enqueue_delay > yy.
```

- Then the formal arguments were added:

```
overloaded_device(D) IF
  rmf_device_specific_start_subchannel_
  rate(D) > XX AND
  rmf_device_specific_enqueue_delay(D) > yy.
```

  We noticed that all of these properties, relations, etc. varied depending on the specific RMF recording interval. A device might be overloaded from 2 p.m. to 3 p.m. but perfectly utilized an hour later. Therefore an implicit argument was identified and added to represent the recording interval:

```
overloaded_device(I,D) IF
  rmf_device_specific_start_subchannel_
  rate(I,D) > XX AND
  rmf_device_specific_enqueue_delay(I,D) > yy.
```

  Because Prolog is a relational rather than a functional language, functions must be expressed as relations. Thus "rmf_device_specific_start_subchannel_rate" is a relationship that holds between a device and a rate during an interval. Other variables representing the resulting rate must be added.

```
overloaded_device(I,D) IF
  rmf_device_specific_start_subchannel_
  rate(I,D,R) AND R > XX
AND
  rmf_device_specific_enqueue_delay(
  I,D,Delay) AND Delay > yy.
```

- The final act of programming for this rule is to specify the user interface information. In this case that simply requires us to tell the application development system that this rule is an interesting part of an explanation about a detected problem. Also, the procedure to be used in producing the explanatory text must be specified. Both things are done by adding another rule that we term a *meta-rule* in the application development system:

```
interesting(overloaded_device(I,D), Text) IF
  concatenate(
  'Device'. D. 'is overloaded in interval'. I, Text).
```

  In this example, the text-formatting procedure "concatenate" produces as second argument a string Text by concatenating each of the strings in the list that is its first argument. Thus, if D is bound to '0570' and I is bound to '13:00,' the above subgoal binds Text to 'Device 0570 is overloaded in interval 13:00.'

  The actual text-formatting rules can be arbitrarily complex, including calling on a natural-language paraphraser if one is available. For this application, relatively simple text formatting—essentially messages with variables—suffices for presenting most results.

System data. For this application, the data base consisted of the RMF monitor data concerning target-system performance and the SYSGEN data concerning target-system configuration. These were data sets found on the MVS system. Because our experimental development environment was VM-based, we had to make the data available in the VM/CMS environment.

Because the data format of the RMF data is not supported on CMS, we wrote a PL/I program to run on MVS that converts the RMF data format to one handled by CMS. At the same time, it normalizes and

reformats the data into the Prolog term structure format. A similar preprocessor was written in Prolog for the SYSGEN data. Subsequent revisions of both programs for MVS/XA provided a dramatic demonstration of the benefit of Prolog. The update time per change for the Prolog program was about one-fifth the time required to update the PL/I program.

Content of the rules is formulated by the expert, and the appearance of the user's view of the application is designed by the programmer/analysts.

Designing the user's view of the application. The general content of the rules is formulated by the expert, and the appearance and structure of the user's view of the application are designed by the programmer/analysts. The following three types of screens are used to interface with the user:

- Menus
- Queries
- Results

Menus, illustrated in Figures 1A, B, and C, give the user choices about what the tuning system is to do. *Queries* are used by the application to get information from the user. A typical query screen for the MVS tuning application is shown in Figure 2. Query screens may in fact be menus in which the user can select an appropriate answer. However, we restrict the use of the term *menu* to designate the first type of screen. *Results* present the sought-for information to the user. An example is shown in Figure 3. Note that the text for "overloaded device" is included in the explanation in Figure 3. Results screens may have any look the author creates. The development environment provides a special service that is used for nearly every result screen in the MVS tuning application. This service provides a file containing the text associated with every interesting rule that went into determining an answer. When one rule invokes another, the hierarchical structure is maintained. The example in Figure 3 has three levels. The result is a file that provides a complete explanation of the results and how they are justified. This file may be displayed as part of a result screen or it may be browsed by the user.

Discussion of results and concluding remarks

Validation. The results of the application have been validated by comparing answers produced by the MVS tuner with answers given by experts for real data in eleven internal IBM installations. The results were accurate for all cases that fell within the scope of the tuner. The system fell short of the expert's ability to extrapolate to new situations. It could not guess about anomalous situations or compare one with another. For example, it could not say why one installation was about half as efficient as another, even though the facilities and workloads seemed similar. The expert could make a guess, because that person was familiar with those systems and had a general knowledge of MVS. The IBM Large Systems Center in Brussels, Belgium, also validated the English form of the rules by reading to see if they made sense and fit the reviewer's concepts of performance tuning. This was not a validation of the program itself, but rather a cross-check on the expertise. The validation showed that although the system performs accurately within its scope, it must be considerably broadened if it is to perform at the level expected of an expert. At a minimum it should be extended to encompass batch, data base, and mixed workloads in addition to TSO environments.

Suitable applications. Our experience with the MVS tuning application suggests that the Prolog language, when augmented with certain application development utilities, is suitable for a large class of applications that have some or all of the following characteristics:

- Are driven via menus
- Engage in a dialog with a user to get particular information
- Embody a significant amount of knowledge about their subject
- Present final as well as intermediate results
- Provide justifications of results
- Require access to shared mainframe data
- Utilize a wide variety of system services
- May be large, small, simple, or complex

The development process for such applications typically requires multiple authors, frequent revision,

and a flexible menu structure. Thus we have found that the method encompasses a wide body of applications. This wide applicability is the result of generalizing from a relatively large application and several smaller ones. At each stage in the development of the tuning application, care has been taken to produce solutions to problems that are expected to be useful in other applications.

The tuner engages in a dialog with the user to get particular information.

The MVS tuning application has all of the previously mentioned characteristics in varying degrees. For example, it is menu-driven. There are three major classes of menus provided in the application, graded according to the amount of performance tuning expertise expected of the user. The tuner also engages in a dialog with the user to get particular information. Since some information is not available from any formal system data, the user is expected to provide such information as special-use devices, groups of operating system users, and the location of such basic information as system trace data.

Within its designed scope, the results of applying the system to actual tuning situations, using 1000 and 4000 rules, show it to be as good as the best expert. The tuning system presents final as well as intermediate results. That is, the system locates performance problems, suggests solutions and their justification, and provides detailed statistics as requested. To do this, the system requires access to large volumes of shared mainframe data. The system trace data that form the bulk of the input to the program consist of complete RMF information detailing such activities as device activity, significant processor activity, paging activity, and operating system users over several recording periods. Because the system handles problems for DASD that are shared among systems, relevant data for all systems connected to a device must be available. The tuning system has access to a wide variety of system services. Through the application development utilities it has access to full-screen mapping, command-language processing, file editing, RMF recording, and system accounting facilities.

In the development process, four authors were involved in the formulation of rules, menus, and ancillary routines. In addition to twice significantly extending the scope of the problems handled by the tuning system, they revised the rules for a new version of the operating system, from MVS/370 to MVS/370 Extended Architecture (MVS/XA).

Problems with the use of Prolog. Like other programming languages, Prolog must provide control constructs, the ability to read and write data, and other nonlogical features. These weaken the declarative nature of a program and emphasize the procedural interpretation. When this happens, the main benefit of using the Prolog language is diluted. The purpose of the tools and utilities we developed is to minimize the need for these nonlogical features. This lets an application author maintain the declarative nature of a program.

The expression of knowledge is but one aspect of developing an application. There are a number of other pragmatic concerns that must be recognized in providing a total development environment, including the following:

- The application must be able to use existing data and programs not originally written to co-operate with the application.
- It must be easy to design and produce the user interface, including screen format and processing.
- Control of the processing of the knowledge rules must be simple and straightforward.
- Debugging aids must be provided.
- Provision must be made for the explanation of results to the user.

Since the Prolog language is so general, there are many ways to express a problem, which is not always an advantage. It gives the author much power in the language, but also requires more understanding and experience than does a less comprehensive language.

Although the *logic* expressed by Prolog clauses is neutral with respect to the processing strategy, Prolog provides only a backward chaining strategy. Other control strategies are sometimes useful.

The power of Prolog's relational data retrieval is available only for data structures represented as re-

lations. The nature of Prolog as a first-order logic subset presupposes that the set of axioms does not change during the proof. This means that the notion of changing state must be explicitly represented as *terms*, which are structured data items that can be passed from procedure to procedure. Examining and modifying these data structures must be explicitly programmed, as in other languages.

Prolog advantageously handles problems whose size allows all relevant data to be loaded into active memory. However, when the problem is too large, one must plan for and program data access on secondary storage (DASD), which Prolog handles with about the same efficiency as such standard languages as PL/I, COBOL, or FORTRAN.

Acknowledgments

The work discussed in this paper was supported by the IBM Data Systems Division. I would like to thank Katherine Arnold, Charles Brotman, and Clarence Poland, who helped put the programming environment for Prolog in place. Ms. Arnold also authored the bulk of the MVS tuning rules. Dave Hunter was very patient in providing us with specific tuning information. Pierre Decoux, of the Large Systems Center, Brussels, Belgium, gave us an independent verification of the rules. I would like to thank James Brady, Charles Head, and Richard Baum for making the work possible.

Cited references

1. A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero, *Un Système de Communication Homme-Machine en Français*, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France (1973).
2. G. Battani and H. Meloni, *Interpreteur du langage de programmation PROLOG*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France (1973).
3. P. Roussel, *PROLOG: Manuel de Reference et d'Utilization*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Marseille, France (1973).
4. D. H. D. Warren and L. M. Pereira, "PROLOG—The language and its implementation compared with Lisp," *ACM SIGPLAN Notices* **12,** No. 8, 109–115 (August 1977).
5. W. G. Wilson, "**PLA** logic programming," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Houston, TX (1980), pp. 897–900.
6. W. G. Wilson and C. John, "Semantic code analysis," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, Karlsruhe, West Germany (August 1983), pp. 520–525.
7. T. Kurokawa, "Logic programming—What does it bring to the software engineering?" *Proceedings of the First International Logic Programming Conference*, Marseille, France (1982), pp. 134–138.
8. A. Walker, "PROLOG/EX1, an inference engine which explains both yes and no answers," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, Karlsruhe, West Germany (August 1983) pp. 526–528.
9. Adrian Walker and Antonio Porto, *KB01, A Knowledge Based Garden Store Assistant*, Research Report RJ-3228, IBM San Jose Research Center, San Jose, CA (1983).
10. L. Pereira, E. Oliveira, and P. Sabatier, "ORBI—An expert system for environmental resource evaluation through natural language," *Proceedings of the First International Logic Programming Conference*, Marseille, France (1982), pp. 200–209.
11. P. Hammond, *Logic Programming for Expert Systems*, Report DOC 82/4, Department of Computing, Imperial College, London (March 1982).
12. E. Oliveira, "Developing expert systems builders in logic programming," *New Generation Computing*, Vol. 2, Springer-Verlag, Inc., New York (1984), pp. 187–194.
13. K. L. Clark and S. Gregory, "A relational language for parallel programming," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture* (1981), pp. 171–178.
14. K. L. Clark and S. Gregory, *PARLOG: A parallel logic programming language*, Research Report DOC 83/5, Department of Computing, Imperial College, London (1983).
15. E. Y. Shapiro, *A Subset of Concurrent PROLOG and its Interpreter*, Technical Report TR-003, Institute for New Generation Computing, ICOT, Tokyo (1983).
16. W. G. Wilson, *PureLog: Pragmatic Logic Programming Using Meta-Declarations*, Ph.D. Dissertation, Syracuse University, Syracuse, NY (May 1985).
17. E. Y. Shapiro and A. Takeuchi, "Object-oriented programming in concurrent PROLOG," *New Generation Computing* **1,** No. 1, 25–48 (1983).
18. M. H. van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the Association for Computing Machinery* **23,** No. 4, 733–742 (October 1976).
19. R. A. Kowalski and D. Kuehner, "Linear resolution with selection function," *Artificial Intelligence* **2,** 227–260 (1971).
20. R. A. Kowalski, *Logic for Problem Solving*, Elsevier-North Holland Publishing Co., New York (1979).
21. R. A. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM* **22,** No. 7, 424–436 (July 1979).
22. J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery* **12,** No. 1, 23–41 (January 1965).
23. C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, Inc., New York (1973).
24. *MVS Extended Architecture Resource Measurement Facility Reference and User's Guide*, LC28-1138, IBM Corporation; available through IBM branch offices.
25. *OS/VS2 System Generation Reference Document*, C26-3792, IBM Corporation; available through IBM branch offices.
26. P. Hammond and M. Sergot, *APES: Augmented Prolog for Expert Systems Reference Manual*, Logic Based Systems Ltd., Richmond, Surrey TW9 2HE, England (1985).
27. P. Hammond, *The APES System: A User Manual*, Report DOC 82/9, Department of Computing, Imperial College, London (September 1982).
28. P. Hammond, *The APES System: A Detailed Description*, Report DOC 82/10, Department of Computing, Imperial College, London (September 1982).
29. *Expert System Development Environment/Consultation Environment General Information Manual*, GH20-9597, IBM Corporation; available through IBM branch offices.

Walter Wilson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.*
Dr. Wilson is a senior programmer in the Symbolic Reasoning Department. He joined IBM in 1970 as a systems programmer in the Data Systems Division in Poughkeepsie, New York, where he was involved in the initial design and implementation of the MVS operating system. He subsequently worked on various interactive facilities for MVS. Prior to his assignment in the Research Division, he was manager of Advanced System Technology in the Data Systems Division Laboratory in Poughkeepsie. Dr. Wilson received his B.S. in mathematics from Stanford University and his M.S. and Ph.D. degrees in computer science from Syracuse University.

Reprint Order No. G321-5271.