NASA Contractor Report 178048

(NASA-CR-178048) IMPLEMENTATION OF ARTIFICIAL INTELLIGENCE RULES IN A DATA BASE MANAGEMENT SYSTEM (VAIR, Inc.) 64 p HC A04/MF A01 CSCL 09B

N86-21220

Unclas

G3/63 05649

Implementation of Artificial Intelligence Rules in a Data Base Management System

Stefan Feyock

VAIR, INC.

Williamsburg, VA 23185

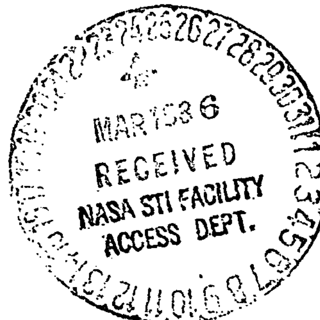
NASA Contract NAS1-18002

February 1986

NASA

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665


MAR 1986
RECEIVED
NASA STI FACILITY
ACCESS DEPT.

PROJECT OVERVIEW

Project Goals

It appears appropriate at the outset to recapitulate the goals of the project. They were as follows:

1. Transform the intelligent front-end prototype into a RIM-integrated system.

2. Write a RIM-based expert system which demonstrates the developed capability.

3. Investigate the use of rules to produce extensibility of the intelligent front end, including the concepts of demons and rule-manipulation rules. Innovative approaches such as syntax programming were to be considered.

All of these tasks were performed successfully; the results form the content of this report. The purpose of the present section is to present an overview of the project's development, leaving presentation of technical detail to subsequent sections.

Paths to the Goals

The facilities available at the beginning of the project for achieving the above goals consisted largely of the LProlog system, a Pascal program created by the author which embodied

list processing facilities similar to LISP, as well as a Prolog implementation based on these list-processing capabilities ("LProlog" denotes "Lisp-Prolog"). The initial decision to be made before the project could progress concerned the question of which facility was to form the intelligent front end: LProlog, Franz Lisp [Foderaro], or the Pascal-based University of York Prolog implementation [Spivey]. After extensive consideration the York Prolog interpreter was chosen. Reasons for this choice include power and efficiency, which gave it the advantage over LProlog, and compactness and linguistic compatibility (Pascal talks easily to other languages and systems), which caused it to preferred to Franz Lisp.

It is interesting to note that hindsight has shown this decision to be largely immaterial: the techniques developed would have led to successful (and analogous) solutions regardless of choice of vehicle. More precisely, it was the concept of the escape predicate that was crucial to the integration of RIM with the chosen front end, and such a function could have been implemented in any of the candidate systems. The chief effect of choosing York Prolog was to allow the creation of extremely concise and elegant programming solutions, as well as a natural integration of RIM as a stream object into the Prolog model. The insight that Prolog is useful and natural but not indispensable has important implications for future work: systems other than Prolog can serve as front end, hence users who are not Prolog programmers can make use of our

results.

As will be seen, even the present Prolog-based system can be made quite friendly to users who have a basic knowledge of Prolog but do not program the language at the expert level. Subsequent sections will discuss the stream and rget procedures, both of which are high-level facilities that are easily understood and can be used to create powerful intelligent database systems. The concept of rules as views allows simple, powerful, high-level constructs to be created from complicated low-level ones. Finally, constructs such as coroutines and demons are constructed on the basis of program templates, which can be filled in by the programmer in a straightforward manner.

The RIM/Prolog system, named YRIM, that was constructed as described was used to create TURBO, a RIM-integrated expert system for jet engine diagnosis. The result is a concise but powerful system whose rules reside in Prolog but which is driven by time-based data stored in RIM. The most salient feature of this program is the complete integration of RIM and Prolog: unless he examines the program, the user has no way of knowing whether the data he is accessing is stored in Prolog's internal database or in RIM. A subsequent section contains a detailed discussion of TURBO.

The final application of our system was to the design of STRUTEX, an expert system for structural design. A design was developed for STRUTEX, as were YRIM programs based on that design; lack of time, however, precluded development of a

running implementation.

Looking Ahead

In light of the clear appropriateness of YRIM for the creation of TURBO, possibly the most interesting result of the STRUTEX exercise was to bring to light a number of areas for which our system appears to be ill-suited. Perhaps the greatest difficulty arises when the appropriate role of the Prolog portion of the system is to act as subroutine rather than main program; Prolog has extreme difficulties accepting such a subordinate role. The idea of using the coroutine capability which was developed was considered as a possible design for STRUTEX, but this solution appeared unduly complicated. We were led back, instead, to the idea presented above: that the techniques developed are widely applicable, and that the use of Prolog as front end is interesting and productive but not essential. The most obvious task for the future is to explore other front ends based on the same interfacing concepts, particularly front ends suited to the requirements of STRUTEX.

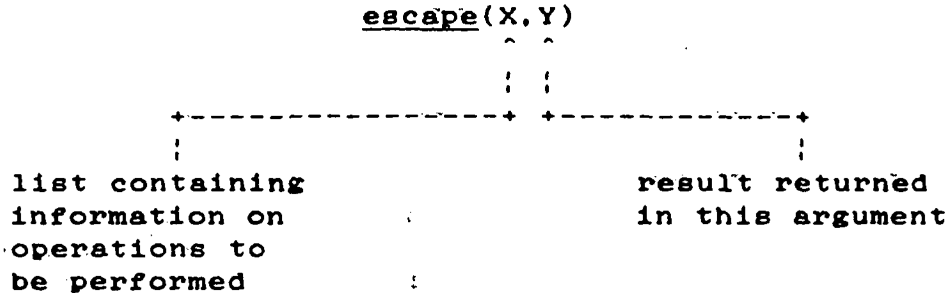
THE PROLOG/RIM INTERFACE

The *escape* Predicate

As has been indicated, the *escape* predicate is the heart of the Prolog/RIM interface; moreover, we have noted that similar predicates can serve as interfaces between a variety of other systems. *escape* would work as well, for example, as a Prolog/graphics package interface, or an LProlog/RIM interface, etc. In fact, the only requirement appears to be lists or list-like structures in the calling language (i.e. the language calling the *escape*), since otherwise the operations needed to set up and decode *escape*'s parameters are too cumbersome. The fact that few languages besides those oriented toward Artificial Intelligence feature list structures as *primitives*, rather than a construct to be defined by the programmer, may account for the fact that the *escape* mechanism we are about to describe is not universal.

In YRIM the *escape* predicate is added to the Prolog side of the interface; it is installed in Prolog as a new evaluable predicate. Details of this installation are given in the IFACE section of the accompanying program documentation.

Here is the design of the escape predicate as it was implemented:



The input list X is expected to be a linear list of atoms (symbolic or numeric); the result appears bound to Y, and also has the form of a linear list of atoms. Note that quoted strings are legitimate atoms in Prolog, so passing a list

[floatadd, '37.82', '-10.036']

is a feasible method of implementing real addition in Prolog. Moreover, while the information that can be passed is quite arbitrary, the format

[<action_code>, <arg>, ---]

is typical.

The interface between Pascal and Prolog consists of a set of procedures within the Prolog implementation that move the values of the input list elements to a parameter buffer internal to the Pascal program on the Pascal side of the interface, whence they may be manipulated by the Pascal program as desired. Returning parameters to Prolog is the reverse of this process: the result values are placed in the parameter buffer, and interface routines use these values to create a Prolog list and bind it to the second parameter of escape. The reader is again referred to

the program documentation for details.

As indicated above, the format [<action_code>, <arg>, ---] is typical for input parameter lists, i.e. parameters to be passed to the *escape* predicate in a list bound to the first parameter. This means that the appropriate format for a Pascal program implementing *escape* is a *case* statement on <action_code>; in other words, the Pascal program is typically an interpreter interpreting commands of the form [<action_code>, <arg>, ---].

As implied by this discussion, the Pascal program that implements the actions taken by *escape* is quite arbitrary, and can contain facilities limited only by the imagination of the user. We have in fact implemented *escape* as a separately compiled module, thus facilitating the addition and modification of *escape* capabilities. This module is included in the RIM_MOD section of the program documentation. For the purposes of YRIM, the *cases* correspond to the RIM interface operations described in the subsequent discussion of *escape*.

The design and implementation of the *escape* evaluable predicate, while requiring a thorough understanding of the mechanisms and data structures underlying the Prolog implementation, was reasonably straightforward. It is therefore rather surprising, in view of the profound implications such a capability has for logic programming, that this feature is not a standard component of Prolog systems; quite the contrary, the author is not aware of any Prolog offering such a capability.

The possibilities opened by this device, however, are startling: Prolog suddenly becomes a vehicle for addressing programming problems for which it could not even be considered before, such as system simulation, operating system design, graphics, and other traditional complex problem areas.

Our interests, of course, lie in the area of databases, and so we will turn to the application of <u>escape</u> to the purposes of the present project. It was decided to take the straightforward approach of implementing an <action_code> corresponding to each of the RIM applications interface routines. For example, to do a

RMFIND(<u>in</u> relation; <u>out</u> cursor)

we simply invoke the predicate escape([1,'YEAR'],C) from Prolog, where 1 is the action code; the cursor (an integer) assigned to relation YEAR is returned in a list bound to C: C = [0], for example. We can then use this cursor to do RMGETs, i.e. to retrieve tuples from YEAR one by one: if X was bound to the cursor found by the RMGET, escape([2,X],Tuple) returns the retrieved tuple as a list bound to Tuple. The RIM_MOD <u>cases</u> corresponding to action codes such as 1 and 2 may be found in the RIM_MOD section of the program documentation.

This capability, of course, leads straight to the use of Prolog to implement the intelligent database front end that is the subject of this project. In particular, it opens the way for the integration of rules into database processing. Before describing how this is done, we first present the complete set of <u>escape</u> action codes that were implemented to form the RIM

interface. It should be noted that the interface refers to the RIM Version 5 implementation available at implementation time, and thus lacks a number of additional features available in RIM Version 6. Most of these are of little significance; one exception is the RIM/6 applications language routine

RMBLD(relname, attname)

that allows the user to specify that the named attribute is a key. This operation is important because the WHERE qualification to the RIM SELECT depends on indices for efficiency. Addition of this capability to the interface is straightforward.

The simplest way to present the implemented primitives is to give annotated transcripts of actual interactive sessions with the YRIM system, in the course of which the newly implemented primitives are exercised.

Here is the RIM database DB accessed by this program:

```
YEAR
        NAME    RANK
        -----------
        FRED     1
        JILL     2
        HENRY    3
        JACK     2


STUDIES
        NAME     CLASSNO
        ---------------
        FRED     611
        JACK     620
        JILL     641
        JILL     646
        HENRY    643
        HENRY    646
```

CLASSES

CLASS	TIME	ROOM
611	M1000	LG1
611	W1300	LG1
646	TU1100	G24
622	TH1000	R418
643	TU1100	R224

Comments in the run will be denoted by Ada-like "--" marks. Unless inappropriate or obvious from context, user input will be denoted by indentation.

```
        run yrim  -- yrim is the Prolog/Pascal/RIM dbms program

   Enter name of file containing Prolog program: prog.rim
        -- prog.rim implements the stream capability
        -- to be discussed subsequently. It will not be needed
        -- here, but yrim insists on reading some Prolog file

        ?- escape([101,db],X). -- uses RMOPEN to open data base db.
        -- recall that the first argument of escape must be a
        -- list, the first element of which is interpreted on
        -- the rim_mod side as an action code. Code 101 denotes
        -- RMOPEN

   ** escape([101,db],[0]) ?; -- Prolog's response. The 0 in the
        -- second (output) parameter is the RMSTAT return code.
        -- 0 denotes successful completion of RIM operation.
        -- The ";" is the user's request for additional
        -- answers, in response to Prolog's prompt of "?"
        -- (i.e. "search for additional answers?")

   no   -- no more answers found, the above was the only one.
        -- We will omit Prolog's responses henceforth
        -- unless they are relevant to the discussion.

        ?- escape([97],X) -- call READSCHEMA
        -- This operation does not correspond to any RIM
        -- operation. Routine READSCHEMA reads the relational
        -- schemas describing the open database into Pascal
        -- arrays for easier access. This is operation is necessary
        -- for the success of subsequent RMGET operations.
```

```
     ?- escape([110,1,studies],X) -- RMFIND(1,'STUDIES')
     -- Result of subsequent RMSTAT is returned in X.

     ?- escape([110,2,year],X) -- RMFIND(1,'YEAR')
     -- On the Prolog side, names of constants must be in
     -- lower-case, while RIM demands that everything be
     -- in capitals. The interface converts automatically.

     ?- escape([99],X). -- show RN_RIX_TABLE
     -- RIM-MOD (the Pascal interface) maintains an internal
     --  table that associates cursor numbers with relation names.

 SLOT #: 0 <FREE>
 SLOT #: 1 <FREE>
 SLOT #: 2 <FREE>
 SLOT #: 3 <FREE>
 SLOT #: 4 <FREE>
 SLOT #: 5 <FREE>

     -- no associations yet

     ?- escape([98,2,year],X) -- associate cursor # 2 with
     -- relation year in RN_RIX_TABLE

     ?- escape([98,1,studies],X) -- associate cursor # 1 with
     -- relation studies in RN_RIX_TABLE

     ?- escape([99],X). -- show RN_RIX_TABLE again

 SLOT #: 0 <FREE>
 SLOT #: 1 STUDIES
 SLOT #: 2 YEAR
 SLOT #: 3 <FREE>
 SLOT #: 4 <FREE>
 SLOT #: 5 <FREE>

     ?- escape([113,1],X). -- RMGET(in cursor_no, out tuple)

** escape([113,1],[studies,fred,611]) ?
     -- tuple <fred,611> has been retrieved from STUDIES.
     -- READ_SCHEMA was needed, since the program must know
     -- about the relations in the database in order to
     -- create a neat answer like  [studies,fred,611]
     -- from the mess RMGET returns.

     ?- escape([113,2],X). -- RMGET(2,X); cursor 2 is on YEAR

** escape([113,2],[year,fred,1]) ?
     -- tuple <fred,1> has been retrieved from YEAR

     ?- escape([113,1],X). -- RMGET(1,X); back to rel. STUDIES
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
** escape([113,1],[studies,jack,620]) ?
      -- tuple <jack,620> has been retrieved from STUDIES

      ?- escape([113,2],X). -- RMGET(2,X); back to rel. YEAR

** escape([113,1],[year,jill,2]) ?
      -- tuple <jill,2> has been retrieved from YEAR

      -- Time to try something different.
      -- The retrieval of tuples from relations produces
      -- tuple streams. RIM also allows the treatment of the
      -- schemata in a database, and the attributes within a
      -- relation, to be treated as streams.

      ?- escape([106],X). -- call to RMLREL, which sets a cursor
      -- to the schema of the first relation in the database.
      -- X returns value of RMSTAT after RMLREL call.

      ?- escape([107],X). -- do a RMGREL
      -- RMGREL(out rname, rpw, mpw, numatt, numrows)

** escape([107],[schema, year, no_rpw, no_mpw, 84/09/28, 2, 4]) ?
      -- format of answer:
      -- [schema, relation name, read password status,
      --  modify password status, date last modified,
      --  number of attributes, number of rows in relation]
      --  The word "schema" is included as flag indicating
      --  that the end of the stream has not yet been reached.
      --  End-of-stream is indicated by an "eoschema"flag
      --  (see below)

      ?- escape([107],X). -- another RMGREL
      -- in order to advance to next schema

** escape([107],[schema, studies, no_rpw, no_mpw, 84/09/28, 2, 6]) ?

      ?- escape([107],X). -- advance to next schema

** escape([107],[schema, classes, no_rpw, no_mpw, 84/09/28, 3, 5]) ?
                  ?- escape([107],X). -- advance again

** escape([107],[eoschema]) ? -- end of schemata reached

   escape([107],X). -- do another RMGREL to see what happens

** escape([107],[schema, studies, no_rpw, no_mpw, 84/09/28, 2, 6]) ?
      -- RMGREL cycles!

      -- Now let us examine the intra-relation cursor created
      -- by RMLATT and advanced by RMGATT
```

```
        ?- escape([108,classes],X). -- RMGATT(in rname)
        -- sets a cursor to schema of named relation

        ?-escape([109],X). -- RMGATT

   **escape([109],[att,class,int, ,novlength,1,0,1,nokey]) ?
        -- Format of answer:
        -- [att, attribute name, attribute type,
        --  matvec: blank unless this attribute is a matrix or vector
        --  var: variable length attribute flag
        --  len1, len2 : see RIM manual [RIM]
        --  column number, key flag]

        ?-escape([109],X).
        -- another RMGATT to advance to next column

   **escape([109],[att,time, text,,novlength,8,0,2,nokey]) ?
        -- descriptor of attribute 2 of relation

        ?-escape([109],X).
        -- advance to next column

   **escape([109],[att,room, text,,novlength,8,0,3,nokey]) ?
        -- descriptor of attribute 3 of relation

        ?-escape([109],X).
        -- advance

   **escape([109],[eoatt ]) ?
        -- end of attributes reached

        ?-escape([109],X).
        -- try it again, see what happens

   **escape([109],[eoatt ]) ?
        -- in contrast to RMGREL, RMGATT does not cycle
```

The above examples illustrate the operation of the implemented operations. A number of additional operations have been implemented; although these are just as important as the above, interaction with them proceeds similarly, and we will therefore confine ourselves to descriptions.

escape([1,<relname>],X) is similar to RMFIND (code 110), but

more "automatic": a cursor number is picked and allocated to relation <relname> by the system, and entered in the RN_RIX_TABLE. X is bound to list [Ok_ix,Cursor,IERR]), where IERR is the result of doing a RMSTAT upon completion.

escape([96,<cursor>],X) undoes code 98 by freeing the named cursor, i.e. removing it from the RN_RIX_TABLE.

escape([112,<cursor>,<attribute_name>,<direction>, ---],X) performs a RMSORT on cursor no. <cursor>. Sorting is on the attribute designated by <attribute_name>.

<direction> ::= asc ¦ desc

specifies whether the sort is ascending or descending. The <attribute_name>,<direction> pair may be repeated, thus specifying sorting on additional attributes. X returns [IERR]. Here is an example of the use of this code:

escape([112,2,rank,asc],X).

Assuming cursor 2 is assigned to relation YEAR, this operation does a RMSORT on attribute rank of relation YEAR, specifying ascending order.

The predicate

escape([111,<cursor>, <num_booleans>,
<attribute_name>, <operator>, <value> ---],X).

implements RMWHER, the most complex of the RIM application language operations. Our implementation simplifies this operation substantially, since most of the effects of RMWHER can be achieved at the Prolog level, and specified far more clearly and elegantly there as well. The significance of RMWHER is that

if an attribute has an index defined on it, then RMWHER can take advantage of this index. This can make the difference between a system confined to toy-sized problems and one that can handle arbitrary relations. Since the RIM implementation of RMWHER restricts this operation to using at most <u>one</u> keyed attribute, it was possible to simplify our implementation by taking advantage of this fact.

As before, the second list element is the cursor number. <num_booleans> is the number of boolean expressions of the form <attribute_name>, <operator>, <value> to follow. <operator> is any of the boolean operators listed in the RIM manual: gt, lt, etc. <value> specifies a value which must be consistent with the type of the attribute. Here is an example:

```
escape([111,2,1,rank,gt,2],X).
```

This operation modifies cursor 2, assumed to be associated with relation YEAR, to produce a stream limited to students of rank > 2. As before, X returns [IERR].

It was noted that performing a RMWHER after a RMSORT did not work (produced an empty stream) at the RIM level, while a RMSORT performed after a RMWHER <u>did</u> work correctly.

Update Operations

The RIM application language interface provides three operations for database update: RMLOAD, which adds a tuple at the end of the designated relation, RMPUT, which modifies the

tuple pointed to by the designated cursor, and RMDEL, which deletes the tuple pointed to by the designated cursor. Clearly a cursor that points to the required tuple must exist for the latter two operations; this cursor is positioned by means of the RMGET operation (code 113).

The predicate

escape([114,<cursor>,<attribute_value>,---],X)

implements RMLOAD. For example, if 2 is allocated to YEAR,

escape([114,2,fred,1],X) will add tuple <fred,1> as new last tuple of YEAR. X returns [IERR].

The predicate

escape([115,<cursor>,<attribute_value>,---],X) implements RMPUT. For example, if 2 is allocated to YEAR,

escape([115,2,fred,1],X)

will replace the tuple to which cursor 2 is currently pointing by the tuple <fred,1>. X returns [IERR].

escape([116,<cursor>],X) implements RMDEL. The tuple pointed to by the designated cursor is deleted from the relation.

Appropriately the last operation to be discussed is escape([102],X), the RMCLOS operation, which closes the current database. Unless this operation is performed, none of the updates performed on the database become permanent.

YRIM Programming

The escape predicates described in the previous section are sufficient to effect all RIM applications language operations from Prolog. It is clear, however, that these predicates are too low-level to be useful for routine programming, whether by a beginner or adept Prolog programmer.

Building on the Primitives

Fortunately Prolog makes the construction of higher-level language elements on the basis of more elementary ones extremely easy and natural. We will illustrate this assertion by showing how the user might do some typical retrievals using the present system, as well as define a view. Additional enhancements to user friendliness include significant improvements in the tracing capability of the Prolog interpreter, as well as a number of additional improvements of a minor nature.

The RIM manipulation facilities based on the escape predicate form a useable but not user-friendly set of operations. To begin a session using a database named TURBO, for example, the user must type

```
escape([101, turbo], X)
```

to open the database, followed by

```
escape([97], X)
```

to cause the schemata describing the relations of the database to be read. A higher-level capability is easily created:

```
open(Dbname) :- escape([101, Dbname], Ierr),!,
                checkstatus(Ierr),
                escape([97],Ierr2),
                checkstatus(Ierr2),!.
```

This operation opens the database whose name is bound to Dbname and readies it by reading in the schemata for use by subsequent operations. The checkstatus predicate can abort processing if the status code returned by the preceding <u>escape</u> operation indicates unsuccessful completion:

```
checkstatus(~1) :-
write(' NO MORE DATA AVAILABLE FOR RETRIEVAL '),!,fail.

checkstatus(0). /* 0 indicates successful completion */

checkstatus(10) :-
 write(' DATABASE FILE DOES NOT CONTAIN A RIM DATABASE '),
 !,fail.
                          ---
checkstatus(111) :-
 write(' MORE THAN 10 RULES PER RELATION '),!,fail.
```

The above definitions simply reproduce the error messages listed in the RIM manual as corresponding to each RMSTAT code. If more specific behavior is desired, it can easily be produced in an analogous manner. For example, in a context in which an end-of-data condition should be reported but not cause subsequent operations to be aborted, a predicate such as

```
eod(-1) :- write(' END OF DATA ENCOUNTERED ').
```

can be defined and used.

The above examples show a typical instance of user-created

high-level database manipulation operations based on the provided _escape_ primitives. It is clear that similar user-friendly operations can be defined for each of the provided _escape_ predicates.

We now turn our attention to a number of additional higher-level constructs, which we believe are sufficiently user-friendly to allow the casual Prolog programmer to use the YRIM system, including the demon mechanisms and coroutines. The result is an environment which, while Prolog-based, is reminiscent of a one-dimensional Query-by-Example system.

Streams

One of the basic ideas of this project was that Logic Programming treated its rulebase as a stream of rules and facts, and that if a database could be made to supply similar streams, then its complete integration into logic programming could be achieved. One of the main results of this project has been the implementation of such a stream capability. Here is the (annotated) Prolog program code that implements this capability.

```
rmfind(Relname,Cursor,Ierr) :- escape([1,Relname],[_, Cursor,Ierr]).
    /*  returns a cursor into relation Relname */

rmget(Cursor, Tuple) :- repeat, escape([2,Cursor],Tuple).
    /*  returns the next tuple from the relation with which
        Cursor is associated, and prompts the user to see if
        he wants additional tuples. This procedure is the Prolog
        image of the RIM application language interface function
        RMGET, and is essentially a "manually operated" stream */
```

```
stream(Rel) :- Rel =.. List, List = [Relname | Relargs],
      rmfind(Relname,C,_), rmget(C,Tuple),
      (Tuple = [e_o_r | _], !,fail ; Tuple = List).
    /*  Manually operated streams are fine, but for actual programming
        we need program-operated streams. The stream procedure
        returns the next tuple of Rel whenever the Prolog program
        executes it */
```

Here is an example of how <u>stream</u> is used to interface Prolog and RIM:

```
year(X,Y) :- stream(year(X,Y)).
```

Upon including this rule in the Prolog program, the user can type queries such as

```
?- year(Student,2).   /* who is a second-year student? */
```

and receive as response

```
** year(jack,2) ? ;
** year(jill,2) ? ;
no  /* no more found */
```

While rather Prolog-flavored in format, this interaction is quite comprehensible; in other words, a suitably restricted set of Prolog procedures can serve as DDL/DML for the layman. The main point to be noted here, however, is that there is no indication at all in a query such as year(Student,2) that RIM is being accessed, rather than Prolog's internal database. The <u>stream</u> predicate thus achieves a seamless integration of Prolog and RIM.

Views

Given <u>stream</u>, several powerful capabilities follow. One of these is the ability to define <u>views</u>. Here is a simple example.

Suppose that we have added the rule

```
studies(X,Y) :- stream(studies(X,Y)).
```

to make the studies relation available to the programmer, and want to find the names of the second-year students who are taking cs646. This corresponds to a join:

```
JOIN year.Students = studies.Students
WHERE year.rank = 2 and studies.class = 646
```

In our Prolog DML the query looks like this:

```
?- year(Student,2), studies(Student, 646).
```

Note the analogy with Query-by-Example: the user is essentially giving the system an <u>example</u> of the kind of response he wants to see. The system responds:

```
** year(jill,2), studies(jill,646) ? ;
**  year(henry,2),  studies(henry,646) ?  ;
no
```

This is clear enough, but somewhat cumbersome, and the user decides he wants to define a view, i.e. a new relation that is defined as the above join. This is done by adding the following rule:

```
new_rel(Student) :- year(Student,2), studies(Student,646).
```

The query

```
?- new_rel(X).
```

then produces the response

```
** new_rel(jill) ? ;
** new_rel(henry) ? ;
no
```

Here is a PROJECT view corresponding to

```
PROJECT CLASSES OVER TIME, ROOM GIVING REL2

rel2(Time, Room) :- stream(classes(_,Time,Room)).
```

The following query poses the question " get all freshmen who are taking graduate courses"

```
?- year(Student,1),studies(Student,Course_no),Course_no > 500.
```

producing the response

```
** year(fred,1),studies(fred,611),611 > 500 ? ;
no   /* no additional solutions were found */
```

It is easy to see from these examples that YRIM provides a powerful and expressive user interface to RIM.

Demons

At this point we have not yet transcended the capabilities of existing database systems (although RIM itself does not offer a view definition facility!). We now show how the interface we have described can be used to integrate rules with data. The underlying idea is as follows: the user should see the rules as being stored in the database as part of the data, regardless of how this effect is actually produced. We have shown, however, that if the stream predicate is used, then the user cannot distinguish between data coming from RIM and data coming from Prolog's own database. In other words, all data looks to the user as if it were stored in the Prolog database, and thus the rules stored in the Prolog database appear to be in the same database as the data actually stored in RIM. We have thus produced a virtual database that has all the RIM operations defined on it (via escape), but intermingles data and rules as

freely as does Prolog. Thus a major goal of this project, the storage of rules in RIM, is achieved in an elegant and efficient fashion.

Once the incorporation of rules into a database has been achieved, numerous powerful capabilities become available. One of these is the inclusion of demons in the database, i.e. of pattern- and data-driven processing that is triggered by database updates. The following Prolog code template shows one method of implementing demons:

```
rget(Query) :-  pre(Query).    /* these are preprocessing demons */
rget(Query) :- stream(Query). /* the relation proper            */
rget(Query) :- post(Query).    /* post-processing demons         */

pre(Query) :-  write(' Begin pre-rule application for  '),
                   write(Query), nl, fail.
/* the fail prevents spurious answers on backtracking */

/*******************************************************/
/* rules to be applied before RIM processing begins,   */
/* if any, here.                                       */
/*******************************************************/

pre(Query) :-  write(' End pre-rule application for  '),
       write(Query), nl, fail.

post(Query) :-  write(' Begin post-rule application for  '),
    write(Query), nl, fail.

/*******************************************************/
/* rules to be applied after RIM processing begins,    */
/* if any, here.                                       */
/*******************************************************/

/* here is an example of such a rule: */

post(studies(Student,cs101)) :- rget(year(Student,1)).
    /* typical demon: all freshmen take cs101 */

post(Query) :-  write(' End post-rule application for  '),
      write(Query),nl, fail.
```

Physically this program segment follows the definition of stream given above. As can be seen, this segment is a template, since it provides slots, indicated by comments, wherein the user may insert demons, or in fact any pre- and post-processing rules he chooses.

The rget predicate is a generalization of the stream predicate: stream is sandwiched between the pre and post predicates. pre and post, in turn, invoke whatever demons the programmer specifies.

The present template specifies as demons a set of self-explanatory printouts telling the user that pre- (post-) processing is beginning (ending). In addition we have included an example of a non-trivial demon, in the form of the rule

```
post(studies(Student,cs101)) :- rget(year(Student,1)).
```

The effect is to add a post-processing demon to the studies relation. The augmented relation is queried as follows:

```
rget(studies(X,Y)).
```

If a more elegant form is desired, the view definition

```
studies(X,Y) :- rget(studies(X,Y)).
```

may be included. The query is then

```
studies(X,Y).
```

The rget(studies(X,Y)) predicate first invokes pre(studies(X,Y)), which merely prints out the corresponding messages. rget then invokes stream(studies(X,Y)), which presents the tuples of the studies relation to the user in the usual

manner. Upon reaching the end of this tuple stream, rget invokes post(studies(X,Y)), which prints out the message

Begin post-rule application for studies(X,Y)

and then activates the rule

```
post(studies(Student,cs101)) :- rget(year(Student,1)).
```

which causes rget to be invoked recursively with year(Student,1)) as argument. This invocation of rget proceeds as before; if any demons had been defined for the year relation, they would be invoked as well. There are none, and so the call

```
rget(year(Student,1))
```

is equivalent to the call

```
stream(year(Student,1)).
```

This is a SELECT on year where year.rank = 1. The effect is to deduce that fred is taking cs101 because he is a first-year student.

While the processing we have described may appear complicated, it should be noted that the user need not concern himself with these details, but needs only to insert the desired demon-defining rules in the indicated template slots. This demon-definition capability, of course, was one of the goals of this project.

Coroutines

A major obstacle to the extensive use of Prolog is its lack of familiar control structures such as if-then-else and while loops (or even goto, for that matter). Backtracking is the only control mechanism provided, and while the effects of the above-mentioned control structures can be "faked" to a large extent by means of backtracking, the resulting constructs can be too obscure and unnatural to permit serious software engineering. The ideal situation (in the opinion of the author) would be to have the main (controlling) program written in some other language such as Pascal, and to call on Prolog only when the service it provides so well is required: deductive database retrieval.

It is unfortunately the case that all major Prolog implementations known to the author (including University of York Prolog) operate as main programs, and could be made to play the role of subroutine only after major overhauls. A number of list-based Prologs exist that <u>can</u> operate as subroutines (including the one implemented by the author using the AI toolkit), but are in general either too toy-like, or too closely tied to a particular LISP implementation, or both. Another approach was required.

The solution that was ultimately adopted retained the original program structure whereby York Prolog is the main program, and communicates with Pascal by means of the <u>escape</u>

predicate that causes invocation of the RIM_MOD Pascal procedure, or more generally of ESC_MOD, the Pascal interpreter of action codes without the RIM specialization. Conceptually, however, Prolog and ESC_MOD are coroutines. Recall that the difference between a main program/subroutine and a coroutine/coroutine relationship is that when a subroutine is called, control is always transferred to the beginning of the subroutine's code, and relinquished at one of the subroutine's <u>return</u> statements (we ignore the unfortunate fact that Pascal has no explicit <u>return</u>), while a call to a coroutine (called a <u>resume</u>) transfers control to the coroutine statement after the one last executed before the coroutine relinquished control to the caller.

In order to achieve the same effect in the ESC_MOD subroutine, it was necessary to maintain a variable <u>resume_pt</u> that kept track of the statement where execution was to resume when control was transferred by the caller. This required the insertion of labels at the resumption points, and a <u>case</u> on <u>resume_pt</u> that transferred control to the appropriate statement within what in reality is a subroutine. The required program transformation leads to a straightforward and stereotyped structure that, despite its aesthetic defects, is quite easy to use. Here it is:

```
procedure coroutine;
label 999,  1000,1001,1002,1003; (* resume labels *)
begin
  case resume_pt of
```

```
  0: goto 1000; 1: goto 1001; 2: goto 1002; 3: goto 1003;
  otherwise
   begin
     writeln(' *** From coroutine: bad resume point: ', resume_pt:1);
     halt;
   end;
  end; (* case *)

1000:
  (* here on first invocation *)
  (* start of computation *)
  writeln; writeln(' ### coprocessor action 1; handing off to Prolog');
  (* end of computation *)

  (* Now set up args to be passed to Prolog *)
  no_of_args := 1;
  set_str_arg(1,'procode1',8);
   (* return string "procode1", length 8, as first (and only) arg *)
  (* End of arg setup section *)

  resume_pt := 1;
  goto 999; (* return , i.e. RESUME Prolog *)

1001:
  (* start of computation *)
  writeln; writeln(' ### coprocessor action 2; handing off to Prolog');
  (* end of computation *)

  (* Now set up args to be passed to Prolog *)
  no_of_args := 1;
  set_str_arg(1,'procode2',8);
   (* return string "procode1", length 8, as first (and only) arg *)
  (* End of arg setup section *)

  resume_pt := 2;
  goto 999; (* return , i.e. RESUME Prolog *)

1002:
  (* start of computation *)
  writeln; writeln(' ### coprocessor action 3; handing off to Prolog');
  (* end of computation *)

  (* Now set up args to be passed to Prolog *)
  no_of_args := 1;
  set_str_arg(1,'procode3',8);
   (* return string "procode1", length 8, as first (and only) arg *)
  (* End of arg setup section *)

  resume_pt := 1;
  goto 999; (* return , i.e. RESUME Prolog *)
1003:;
999:; end; (* coroutine *)
```

As it happens, this routine is only one of the services invocable from Prolog by means of action codes; all other action codes are still available as well. Programs that call each other as coroutines, even if only conceptually, are equals, and essentially treat each other as objects. This implies that ESC_MOD can call on the Prolog program for services just as the Prolog program can call on ESC_MOD, and thus the Prolog program will in general have a special structure as well. Here is an example of a structure that can cooperate with the above program:

```
 cycle(_) :- To_outside = [init],
 repeat, communicate(To_outside, From_outside).
    /* dummy vble is there since it won't prompt for more if
goal is a ground clause */

communicate(To,From) :- escape([0,To],From), execute(From,To),!.
  /* 0 is escape coroutine code.
      From should have format [prolog_action_code | arg_list]
      To is result list, possibly same format
 */

execute([procode1],X) :-
write(' Prolog action 1; handing off to escape '),nl.

execute([procode2],X) :-
   write(' Prolog action 2; handing off to escape '),nl.

execute([procode3],X) :-
   write(' Prolog action 3; handing off to escape '),nl.
```

The output of these cooperating programs is left as an exercise for the interested reader.

We have described the interface between Prolog and RIM, and its use in the construction of virtual databases that contain

rules, demons, views, and in fact any arbitrary extensions and capabilities the user cares to program. We now turn our attention to the application of these concepts to the achievement of a main goal of this project: construction of RIM-based expert systems.

TURBO

A RIM-based Expert System

A major goal of this project has been the creation and demonstration of an expert system capability based on the RIM database system. It was decided to use a Prolog-based expert system which has fault diagnosis for a turbofan jet engine as its domain of expertise [Abbott] as a test subject. We will present excerpts of significant portions of her Prolog-based system, as well as their implementation in our RIM-based system.

Both the original system and our YRIM version are driven by data from a predicate named <u>sensed</u>, which records engine sensor readings along with their values and the time (measured in unit increments) the reading arrived. The predicate in Abbott's implementation is

```
sensed(thrust,decreasing,1)
sensed(vibration,yes,1)
sensed(egt,fluctuating,1)
sensed(epr,fluctuating,2)
sensed(n1,fluctuating,2)
sensed(n2,fluctuating,2)
sensed(fuelflow,high,4)
sensed(egt,decreasing,5)
sensed(epr,decreasing,5)
sensed(n1,decreasing,5)
sensed(n2,decreasing,5)
sensed(fuelflow,decreasing,5)
sensed(thrust,decreasing,5)
```

Our YRIM implementation is driven by the corresponding RIM

relation:

Relation SENSED:

SIGNAL	VALUE	TIME
THRUST	DECREASING	1
VIBRATION	YES	1
EGT	FLUCTUATING	1
EPR	FLUCTUATING	2
N1	FLUCTUATING	2
N2	FLUCTUATING	2
FUELFLOW	HIGH	4
EGT	DECR	5
EPR	DECR	5
N1	DECR	5
N2	DECR	5
FUELFLOW	DECR	5
THRUST	DECR	5

Similarly, the RIM relation

CANCAUSE

CAUSE	EFFECT
FOREIGN_OBJECT	COMPRESSOR_STALL
ICING	COMPRESSOR_STALL
COMPRESSOR_STALL	FLAMEOUT
ICING	FLAMEOUT
TURBULENCE	FLAMEOUT
FOREIGN_OBJECT	FLAMEOUT
FUELLEAK	FLAMEOUT
BAD_FUEL_CONTROL	FLAMEOUT

appears to the user as the Prolog relation cancause(Cause,Effect), given a view definition for cancause:

```
cancause(Cause,compressor_stall)
    :- stream(cancause(Cause,compressor_stall)).
```

Even this simple database allows some interesting questions to be answered, such as

```
SELECT CAUSE FROM CANCAUSE WHERE EFFECT EQ COMPRESSOR_STALL
```

which retrieves the causes of compressor stall: foreign objects

and icing. The formulation of this query in YRIM is

```
cancause(Cause,compressor_stall).
```

The SUPERSEDes relation contains information about what faults subsume others. For example, the set of symptoms that indicate a flameout are a subset of the symptoms that indicate a compressor stall. The RIM relation

SUPERSED

SUPER	SUB
COMPRESSOR_STALL	FLAMEOUT
COMPRESSOR_STALL	BAD_FUEL_CONTROL
COMPRESSOR_STALL	TURBINE_SEPARATION

becomes the (virtual) Prolog predicate supersedes(Super, Sub), defined by the rule

```
supersedes(Super, Sub) :- stream(supersed(Super, Sub)).
```

<u>supersedes</u> thus appears to the user as if it were defined by Abbott's Prolog predicates

```
supersedes(compressor_stall,flameout)
supersedes(compressor_stall,bad_fuel_control)
supersedes(compressor_stall,turbine_separation)
```

Note that the name of the RIM relation differs from the virtual predicate defined on the Prolog side, since RIM confines relation names to eight characters. For the same reason, a number of the table entries have been 'de-abbreviated' for readability.

The Turbo Progrem

We will now discuss the TURBO expert system in detail. The program consists of three logical sections; we have marked these divisions in the program. We first present the program as a whole, and then discuss each logical section in turn.

```
/*******************************************************/
/*                    BEGIN PROGRAM                    */
/*******************************************************/

rmfind(Relname,Cursor,Ierr) :- escape([1,Relname],[_, Cursor,Ierr]).

rmget(Cursor, Tuple) :- repeat, escape([2,Cursor],Tuple).

stream(Rel) :- Rel =.. List, List = [Relname | Relargs],
      rmfind(Relname,C,_), rmget(C,Tuple),
      (Tuple = [e_o_r | _], !,fail ; Tuple = List).

dbstart(Dbname,Ierr) :- escape([101,Dbname],Ierr).

readschema(Ierr) :- escape([97],Ierr). /* -1 means: no more data */

turbo :- dbstart(turbo,Ierr),!,Ierr = [0],!,readschema(Jerr),!,
        Jerr = [~1],!, nl,write('  OK, turbo has been read'),nl.

rmlrel(I) :- escape([106],I).

rmgrel(X) :- escape([107],X).

schemata(X) :- rmlrel(I),repeat,rmgrel(X).

?-turbo. /* start up turbo db. The "?-" formulation causes
turbo to be activated when the program is loaded */

/*******************************************************/
/*                   END SECTION 1                     */
/*******************************************************/
```

```
/*****************************************************/
/*                  BEGIN SECTION 2                  */
/*****************************************************/


mostrecent(Signal,Value,Timesensed,Currtime) :-
    not(var(Currtime)),
    rmfind(sensed, Cursor, Ierr), /* allocate a cursor */
    escape([111,Cursor,1,time,le,Currtime],X),
     /* where attr. time <= Currtime */
    escape([112,Cursor,time,desc],X),  /* sort down */
    rmget(Cursor,Tuple),
    (Tuple = [e_o_r | _],!,escape([96,Cursor],_),fail ;
     Tuple = [sensed,Signal,Value,Timesensed],!,escape([96,Cursor],_)
    ).
    /* Timesensed is an output parameter, Currtime is input */

holds(Signal, Value, Currtime)
    :- mostrecent(Signal, Value,Timesensed, Currtime),!.

holds(Signal, X, Currtime) :- var(X), X = normal.
/* It is meaningless to call holds(S,normal,T), since "normal"
   never appears as value in SENSED */

/*****************************************************/
/*                  END SECTION 2                    */
/*****************************************************/


/*****************************************************/
/*                  BEGIN SECTION 3                  */
/*****************************************************/

supersedes(X,Y) :- stream(supersed(X,Y)).
/* supersed is RIM's name for this relation, so we are confined
   to eight characters. Prolog does not require us to be so
   terse. */

cancause(X,Y) :- stream(cancause(X,Y)).

 /* The morelikely predicate expresses the fact that a
  fault is more likely if some conditions or faults that
  can cause it are present at a prior time. */

morelikely(Fault, X, Time) :-
    poss(Fault, Time), cancause(X,Fault), poss(X, Time).

morelikely(Fault, X, Time) :-
    poss(Fault, Time),supersedes(Fault,X),poss(X, Time).

/* poss(Fault, T) being true means that Fault is a possibility
   at time T */
```

```
poss(for_obj, T) :-
  holds(vibrat,yes,T), holds(egt,fluct,T), holds(epr,fluct,T),
  holds(n1,fluct,T),holds(n2,fluct,T),
  ((holds(fuelflow,X,T),X = normal);holds(fuelflow,fluct,T)).

  /* The fault names appear in abbreviated form here, e.g.
     "for_obj" instead of "foreign_object". The abbreviated form
     is what is actually used both in the program and the RIM
     database, since this simplifies the interface with RIM */

poss(cprstall,T) :-
  holds(vibrat,yes,T),
  holds(egt,X,T), X \= normal,
  (holds(epr,fluct,T) ; holds(epr,decr,T)),
  (holds(n1,fluct,T) ; holds(n1,decr,T)),
  holds(n2,fluct,T),
  ((holds(fuelflow,Y,T),Y = normal) ; holds(fuelflow,fluct,T)),
  ((holds(thrust,Z,T),Z = normal) ; holds(thrust,decr,T)).

poss(turb_separation,T) :-  /* turbine blade separation */
  holds(vibrat,yes,T),
  holds(epr,decr,T),
  holds(thrust,decr,T).

poss(icing,T) :-
  holds(oat,X,T), X \= normal, X =< 8,
  holds(moisture,visible,T),
  (holds(vibrat,yes,T) ; holds(epr,fixed,T)).

poss(flameout, T) :-
  holds(egt,decr,T), holds(epr,decr,T),
  holds(n1,decr,T),holds(n2,decr,T),
  holds(fuelflow,decr,T),holds(thrust,decr,T).

poss(fuelleak,T) :-  holds(fuelflow,high,T).

poss(fuelleak,T) :-
  holds(n1,low,T),
  holds(fuelflow,low,T).

/*********************************************************/
/*                    END PROGRAM                        */
/*********************************************************/
```

Discussion of the Program

Section 1 of the program consists of low-level functions that handle the RIM interface. The following rules form the familiar but crucial definition of the stream predicate, which has been discussed in a previous section.

```
rmfind(Relname,Cursor,Ierr) :- escape([1,Relname],[_, Cursor,Ierr]).

rmget(Cursor, Tuple) :- repeat, escape([2,Cursor],Tuple).

stream(Rel) :- Rel =.. List, List = [Relname | Relargs],
      rmfind(Relname,C,_), rmget(C,Tuple),
      (Tuple = [e_o_r | _], !,fail ; Tuple = List).
```

The following rules illustrate how easily a friendlier interface can be constructed from the primitives. dbstart, for example, opens the named database and returns the status code in Ierr. readschema, as its name indicates, reads a (the next) schema from the currently open database.

```
dbstart(Dbname,Ierr) :- escape([101,Dbname],Ierr).

readschema(Ierr) :- escape([97],Ierr). /* -1 means: no more data */
```

Proceeding to the next higher definitional level, we use the previously defined predicates to construct the third-level interface predicate turbo, which opens and initiates the database named TURBO:

```
turbo :- dbstart(turbo,Ierr),!,Ierr = [0],!,readschema(Jerr),!,
        Jerr = [~1],!, nl,write(' OK, turbo has been read'),nl.

?-turbo. /* start up turbo db. The "?-" formulation causes
turbo to be activated when the program is loaded */
```

This ends the first part of the program. The second section defines the crucial mostrecent predicate, which is used in defining the holds predicate, the basic vehicle for defining the

faults themselves.

The *holds* and *mostrecent* Predicates

We now turn to Section 2 of the TURBO program, which defines the crucial predicate *mostrecent*, which in turn defines *hold*. The definition of holds is a *view*:

```
holds(Signal, Value, Currtime)
    :- mostrecent(Signal, Value,Timesensed, Currtime),!.

holds(Signal, X, Currtime) :- var(X), X = normal.
/* It is meaningless to call holds(S,normal,T), since "normal"
   never appears as value in SENSED */
```

Intuitively, holds(S,V,T) is true iff signal S has value V at time T. It is possible, however, that there is no tuple in the SENSED relation for <S,V,T>. Suppose, for example, that T = 48, S = "egt", and V = "fluctuating", but that the only tuple in SENSED for which SENSED.Signal = "egt" and SENSED.Value = "fluctuating" is <egt,fluctuating,3>. Since egt was sensed to be fluctuating at time 3, and no subsequent sensor reading refers to any different sensing of egt, it must be assumed that the egt is still fluctuating at time 48. As we will see, the *mostrecent* predicate insures that *holds* behaves in the required manner.

If no sensor reading at all exists for a particular signal, it is assumed to be normal; the second part of the definition of *holds* corresponds to this requirement.

mostrecent

The predicates discussed heretofore are similar to their counterparts in Abbott's Prolog implementation; it is the mostrecent predicate that forms the interface with RIM, and thus differs completely from its Prolog version. mostrecent retrieves the most recently (relative to a given time, usually the present) sensed value for a signal, and the time at which it was sensed.

Prolog's deficient control structures force the Prolog programmer into a series of clumsy and inefficient constructs in order to implement the following loop:

```
   for row := last row added downto first row added do
    if sensed[row] contains the sought attribute values
    then return row;
```

The RIM implementation proceeds differently:

```
 mostrecent(Signal,Value,Timesensed,Currtime) :-
 /*1*/  not(var(Currtime)), /* Current time must be an input vble */
 /*2*/  rmfind(sensed,Cursor,Ierr), /* allocate a cursor */
 /*3*/  escape([111,Cursor,1,time,le,Currtime],X),
         /* where attr. time <= Currtime */
 /*4*/  escape([112,Cursor,time,desc],X),
 /*5*/  rmget(Cursor,Tuple),
 /*6*/  (Tuple = [e_o_r ¦ _],!,escape([96,Cursor],_),fail;
 /*7*/   Tuple = [sensed,Signal,Value,Timesensed],!,
 /*8*/            escape([96,Cursor],_)
        ).
```

Since this sort of interaction between a database and an expert system appears to be unique, we will discuss this definition in detail.

Line 1: variable Currtime represents the time at which the diagnosis is made. It is usually, though not necessarily, the current time. It is thus an input variable which must be

instantiated when mostrecent is invoked; since the var predicate is true iff its argument is not instantiated, line 1 has the desired effect.

Line 2: The rmfind predicate has been discussed in a previous section; we recapitulate it here:

```
rmfind(Relname,Cursor,Ierr)
     :- escape([1,Relname],[_,Cursor,Ierr]).
```

As can be seen, rmfind invokes escape with code 1, which performs a RIM RMFIND operation on the named relation, and returns the resulting cursor ("pointer", in RIM terms) in output variable Cursor. Ierr returns the resulting error code. The effect of line 2 is thus to allocate a cursor to relation SENSED, and to return this cursor as the value of variable Cursor.

Lines 3 and 4: these two lines perform the essential part of the required retrieval. Line 3 modifies the cursor allocated by line 2 according to the following WHERE clause:

WHERE SENSED.TIME LE CURRTIME

so that all tuples corresponding to values sensed after Currtime become invisible to this cursor.

Line 4 corresponds to the RIM clause

SORTED BY SENSED.TIME = DESCENDING

which has the effect that the first tuple retrieved by this cursor has the largest value in its TIME attribute, i.e. is the most recent value sensed.

Line 5 performs the actual retrieval by effecting a RMGET using Cursor as modified by the above WHERE and SORTED clauses;

line 6 is the "failure" branch taken in case no tuple was retrieved (in which case the signal value is presumably normal), while line 7 converts the retrieved tuple to the appropriate format, given a successful retrieval. These lines are unnecessarily obscure and "Prology" due to Prolog's lack of simple amenities such as an if-then-else.

Finally, line 8 deallocates Cursor, a necessary operation in view of the fact that each invocation of <u>holds</u> requires its own cursor.

It is worth reemphasizing that while the occurrence of actual escape clauses in Prolog/RIM programs is certainly permissible, it is never <u>necessary</u>. If such clauses are deemed to make the program too obscure and unreadable, they can always be replaced by more user-friendly constructs. For example, the clause

```
escape([96,Cursor],_)
```

could be replaced by, say, free(Cursor), and the following definition be added:

```
free(Cursor) :- escape([96,Cursor],_).
```

We have not done this in our example, partly because of time constraints, and partly because such specialized usages of <u>escape</u> clauses did not appear to merit inventing "friendly" forms that would never be used elsewhere.

Fault Diagnosis Predicates

The third section of the program contains the definitions of the fault diagnosis predicates themselves. As we have seen, supersedes and cancause are simply defined in terms of the corresponding RIM relations; the user is unaware whether supersedes and cancause access Prolog's internal database or RIM.

The morelikely predicate uses supersedes and cancause to express the fact that a fault is more likely if some conditions or faults that can cause it are present at a prior time.

The central predicate of the TURBO system, both in Prolog and in our RIM-based system, is the poss ("possible") predicate. poss relates diagnoses to symptoms which may have occured at the present time or earlier. It is this time-oriented approach, unusual in expert systems, which accounts for the complexity of the mostrecent predicate. The reader is invited to inspect this predicate closely, since it is a perfect example of Prolog at its best: these rules are readable and concise, but specify a large amount of information, and even more computation.

The usual question posed to TURBO has the form

```
poss(X,2). /* what diagnoses are possible, given the symptoms
              present at time T = 2? */
```

poss(D,T) is true iff diagnosis D is a possible fault diagnosis at time T. The time parameter generally represents the current time, though any arbitrary value may be used as well. The possible diagnoses are returned as instantiations of

the query variable X:

```
poss(foreign_object, 2) ? ;
poss(compressor_stall, 2) ? ;
no   /* no more diagnoses */
```

Here is a sample run of TURBO. User inputs follow the ?- prompts; also, the ";" following system-generated solutions are user requests for additional solutions.

```
  OK, turbo has been read
?- poss(X,1).
no
/* symptoms at time 1 do not yet confirm any diagnosis */

?- poss(X,2). /* what diagnoses are possible at time = 2? */
** poss(foreign_object, 2) ? ;
** poss(compressor_stall, 2) ? ;
no
/* foreign_object and compressor_stall are the (only)
   possible faults at time = 2 */

?- poss(X,3). /* what about time = 3? */
** poss(foreign_object, 3) ? ;
** poss(compressor_stall, 3) ? ;
no
/* no new symptoms at time 3, so no change */

?- poss(X,4).
** poss(fuel_leak,4) ? ;
no
/* New symptom at time 4: FUELFLOW HIGH. This precludes the
   previous diagnoses of foreign_object and compressor_stall,
   both of which require normal fuel flow. This is a typical
   example of the important role played by time in this
   system: earlier data is made obsolete by subsequent data.*/

?- poss(X,5).
** poss(turb_separation, 5) ? ;
** poss(flameout, 5) ? ;
** poss(fuel_leak, 5) ? ;
no
/* Lots of new symptoms at time 5. This plane has real
   problems. */

?- poss(X,6).
** poss(turb_separation, 6) ? ;
** poss(flameout, 6) ? ;
```

```
** poss(fuel_leak, 6) ? ;
no
/* No new symptoms after 5. */

?- morelikely(X,Y,5).
** morelikely(flameout, fuel_leak, 5) ? ;
no
/* The likelihood that we have a flameout at time 5 is
   increased by the fact that we also had a fuel leak at
   or before time 5 */

?- end.
```

TURBO: Summary

We have used the YRIM system to construct TURBO, an expert system for jet engine diagnosis. The result has been a program that is extremely concise, but at the same time surprisingly powerful: the time dimension is handled appropriately, and the tendency of faults to cause, or at least increase the likelihood, of other faults is represented. Most importantly, the actual fault data is stored in RIM, a fact which is transparent to the user. It is interesting to note that the RIM retrievals involved in constructing the mostrecent predicate are intuitively simpler than the corresponding Prolog program accessing a Prolog database.

The effectiveness of our system for the construction of powerful RIM-based expert systems thus appears to be proven.

STRUTEX

The previous section discussed the creation of the RIM-based expert system TURBO. It was noted that Prolog/RIM appeared eminently suited to this application. Since then we have been supplied by the project monitor with a second expert system application, which has yielded a number of interesting insights and results. This section is largely focussed on the problems posed by this application.

The application in question, which has been named STRUTEX, is an expert system application intended to assist the user with structural design. Time limitations, as well as the fluid and rapidly developing nature of the STRUTEX system specifications, precluded production of an implementation. We present here a design which, although based on a preliminary formulation of the problem, appears to be appropriate for such a system.

STRUTEX is of interest for two reasons. First, its structure is typical, and thus solutions developed for STRUTEX may be expected to carry over to larger systems. This consideration is particularly significant in light of the fact that the design which was developed for STRUTEX is based on close interaction between RIM and the controlling language (presently Prolog), and that the expert system rules are stored in RIM itself. Second, it was noted that YRIM was considerably less well-suited to the

STRUTEX application than to the implementation of TURBO. Several reasons for this situation exist, chief among which is the fact that YRIM uses Prolog as its main program, while STRUTEX requires the expert system to play the role of subroutine. Prolog, however, is ill-suited to such a subordinate role, and alternate approaches had to be developed.

Here are the rules that comprise the preliminary version of STRUTEX available at the time this problem was first addressed:

1. IF SL = 1 AND TL = 1 THEN SOLUTION = 1
2. IF SL = 1 AND SA = 1 AND TL IN {2,3,4,5} AND WC = 2 THEN SOLUTION = 2
3. IF SL = 1 AND SA = 2 AND TL IN {2,3,4,5} AND WC = 1 THEN SOLUTION = 2
4. IF SL = 2 AND SA = 1 THEN SOLUTION = 4
5. IF SL = 2 AND SA = 2 AND WC = 2 THEN SOLUTION = 4
6. IF SL = 2 AND SA = 2 AND WC = 1 THEN SOLUTION = 5
7. IF SL = 3 AND SA = 1 THEN SOLUTION = 6
8. IF SL = 3 AND SA = 2 AND WC = 2 THEN SOLUTION = 6
9. IF SL = 3 AND SA = 2 AND WC = 1 THEN SOLUTION = 7

We have used the following abbreviations in these rules:

SL: structural load. Values of 1, 2, and 3 represent the cases where the support is respectively above, beside, or below the load.

SA: support area (as fraction of distance between load point and support. SA = 1 (2) denotes that this ratio is less (greater) than 0.05.

TL: type of load. Values of 1, 2, 3, 4, 5 represent gravity load, alternating load, side load, or combinations of 1, 3 and 2, 3 respectively.

WC: structure weight. WC = 1 denotes that the structure must be as light as possible; WC = 2 indicates that weight is not important.

Abbreviations such as "SL = 3" would, of course, be translated into a human-oriented (possibly graphic) form before presentation to the user.

The above rules are easily represented as a syntax program. [Feyock]. Rule 1, for example, becomes

1. <solution=1> ::= <sl=1> <tl=1>

while rule 2 translates to

2. <solution=2> ::= <sl=1> <sa=1> <tl=2345>

<tl=2345> ::= <tl=2> ¦ <tl=3> ¦ <tl=4> ¦ <tl=5>

The resulting syntax program could then be fed to an appropriate parser generator, which would automatically produce the corresponding expert system. For the present application, this course of action was deemed to constitute overkill, as well as making the interface of the expert system with RIM unduly complicated. It was therefore decided to perform the grammar-to-expert system conversion by hand, and, in particular, to represent the grammar in transition diagram form before conversion. Transition diagrams have a number of desirable properties that make them appropriate interfaces to databases. Moreover, transition diagrams are closely related to BNF specifications; in fact, any grammar all of whose productions have the form

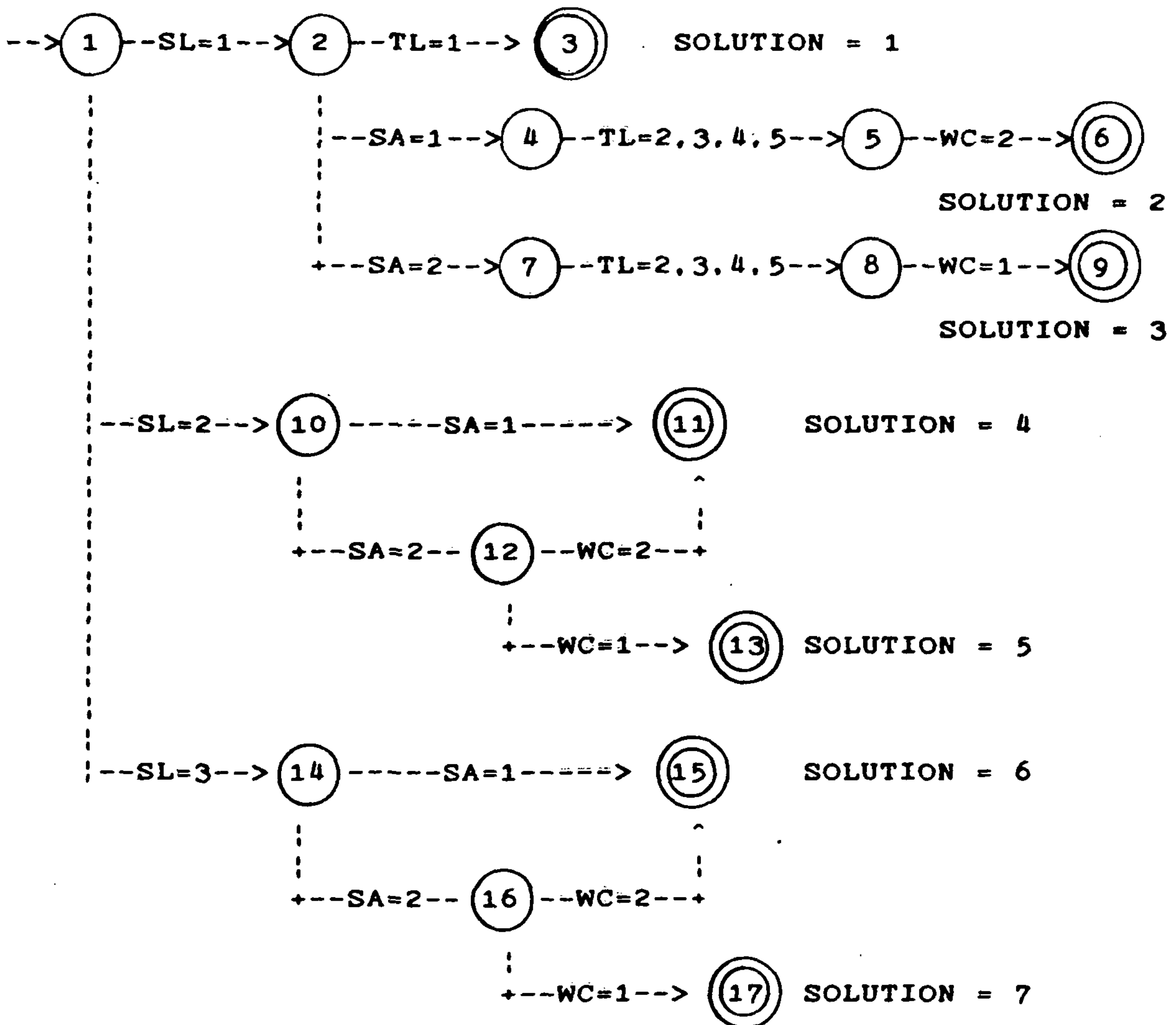
<non_terminal> ::= <non_terminal> terminal

or

<non_terminal> ::= terminal

specifies a transition diagram; conversely, any transition diagram can be specified in terms of such a grammar. Moreover, given an arbitrary BNF grammar (such as the above syntax program), a transition diagram grammar specifying the same system is easily found. Details of this representational equivalence may be found in a number of sources, such as [Gries].

The representational conversions just described were performed on the STRUTEX rule set. Here is the resulting transition diagram:



The most significant feature of the transition diagram (td) representation for expert systems is the fact that most of the information specifying a td can be stored in a relational database in a straightforward manner. The behavior of such a system can be arbitrarily complex; for purposes of illustration, however, the preliminary version of the STRUTEX system we have been discussing will serve well. Consider what the system's

behavior must be when in an arbitrary (non-final) state such as state 2. First, the user must be prompted for an appropriate response. In the context of the STRUTEX application such a prompt is most appropriately presented in menu form:

WHICH ONE OF THE FOLLOWING IS TRUE:

1. TL = 1
2. SA = 1
3 SA = 2

TYPE THE NUMBER OF YOUR CHOICE

It is understood throughout this discussion, of course, that locutions such as "TL = 1" are used solely as abbreviations for what would be human-oriented utterances such as

1. TYPE OF LOAD IS GRAVITY LOAD ONLY

The user types his response (we ignore details such as processing erroneous responses here), and the system transits to the appropriate next state. Upon entering any state, the system must check whether the new state is a final state; if so, the message appropriate to that state (which would presumably be the final diagnosis) is printed and td processing terminated.

From the above description we can determine the information that must be stored in the relations describing the td. Here is a possible database design:

TD	FROM_STATE	MENU_CHOICE	TO_STATE
	2	1	3
	2	2	4
	2	3	7
	4	1	5

	10	1	11
	10	2	12
		etc.	

The TD relation represents the arrows present in the td. The menu messages associated with each state must also be represented:

MENUS	STATE	MENU_CHOICE	MESSAGE
	2	1	TL = 1
	2	2	SA = 1
	2	3	SA = 2

As indicated above, the menu messages stored in the MESSAGE column are not literally "TL = 1", but rather are user-oriented text strings of arbitrary length. This fact raises an interesting problem which at first glance appears to be merely a matter of fixing up the Prolog/RIM interface, but is actually representative of a more fundamental principle which had not been properly recognized heretofore. The problem arises as follows: the MESSAGE attribute is appropriately of type

TEXT(VARYING), allowing arbitrary-length character strings to be stored. The Prolog/RIM interface, however, makes provision only for type TEXT(8). The first impulse upon noting this difficulty is to modify the Prolog/RIM interface to accept text strings of arbitrary lengths. It soon becomes apparent, however, that this modification deals with only the tip of the iceberg: closer examination discloses a host of RIM types for which similar modifications must be made. This task is tedious and produces code that is not only unaesthetic, but also quite RIM-specific. It was the goal of this project from the beginning, however, to develop concepts that would work well with RIM, but would have general applicability as well.

The above considerations led to a reappraisal of the YRIM interface design. It became apparent that the role of Prolog in this design was to provide control structures that allowed AI programming to be done on the basis of RIM. To permit such programming it is necessary for certain information to pass back and forth between RIM and Prolog (or whatever other language implements the superstructure). It is not necessary, however, that all information present in either system be transmittable to the partner; all that is needed is that the control information that directs the other partner what to do with the information in question be transmittable. To take a concrete example, it is not necessary to transmit text strings of arbitrary length from the MESSAGE relation in RIM to Prolog for subsequent presentation to the user. All that is required is

that Prolog be able to transmit to RIM the control information that directs RIM to present the text strings in question to the user.

This concept of passing control information rather than data back and forth between the superstructure language and the database system is expected to greatly increase the power and flexibility of our system, and is an appropriate topic for future research. For the present we return to the presentation of the STRUTEX td.

It is our intent to begin with the simplest possible implementation of the STRUTEX td, and to defer the addition of bells and whistles. It was also considered desirable to investigate questions regarding which side of the Prolog/RIM interface should store which information. It was decided that for the initial implementation the TD relation should reside in RIM, while the information displayed as stored in the MENUS relation was to be stored and manipulated as part of the Prolog database. As it happened, our design makes these considerations transparent not only to the user but also to the Prolog programmer: the information is sought in the Prolog database first; RIM is accessed only if this seek is unsuccessful. Here is a draft version of our td-based expert system "runner":

```
dotransition :-
      retrieve(currentstate(/* out */ Currstate)),
      /* retrieve performs a database access as described */
      /* relation currentstate is currently maintained
         on the Prolog side */
      do_action(/* in */ Currstate, /* out */ Response),
      retrieve(td(/* in */ Currstate, Response,
                  /* out */ Nextstate)),
      /* this currently accesses RIM relation TD */
      update(currentstate(/* in */ Currstate),   /* to */
             currentstate(/* in */ Nextstate)).
      /* update updates the referenced relation */

/* conceptually currentstate is a one-row, one-column
   relation that always contains the current td state */

currentstate(/* state */ 1).
/* State 1 is the start state */

do_action(/* in */ State, /* out */ Response)
      :- menu(State), read(Response).

menu(/* state */ 1) :-
      write(' Welcome to STRUTEX '), nl, /* new line */
      write(' Here are your options: '), nl,
      write(' 1. SL = 1 '), nl,
      write(' 2. SL = 2 '), nl,
      write(' 3. SL = 3 '), nl,
      write(' Type the number of your choice: ').
      /* Could ultimately actually draw a graphic of the
         referenced structural layout */

/* Here is the database access procedure: */

retrieve(Query) :- Query, !. /* look in Prolog db first */

retrieve(Query) :- stream(Query). /* then look in RIM */

/* The stream procedure has been described in an earlier
   section. That's really all there is to it! */
```

The short procedure that implements the loop

repeat dotransition until Currstate in finalstates

has been omitted, since Prolog's lack of such control structures forces the use of inelegant expedients to produce the looping

effect.

Although the expert system implemented by the above td is only a small prototype, the simplicity and elegance of the database access procedure retrieve is striking, and provides evidence of the soundness of the original interface design.

YRIM Programming: Summary

Several important points have been illustrated by the above programs:

1. YRIM completely integrates RIM and Prolog. Our system not only communicates with RIM efficiently, but allows RIM operations to appear in a form identical to Prolog operations. It is thus not only possible but easy to define operations that make it impossible for the user to tell whether he is accessing data stored in Prolog's internal database, or in RIM.

2. The resulting programs are powerful. The use of a Prolog-based system allows queries to be posed that transcend those answerable by rule-based systems based on less powerful languages.

3. The programs written using this system are extremely concise. This point has been in evidence programs described in previous sections, particularly TURBO and the dotransition program described above. In view of the power of these short programs it is clear that this brevity is a result of the expressive power of the underlying system rather than the role

of these programs as demonstration vehicles: programs using this system will *always* be significantly more concise than programs expressed by other means.

CONCLUSION AND FUTURE RESEARCH

It is time to step back from the welter of technical detail that has been described in order to obtain an overview of what has been accomplished. As has been indicated, a seamless integration of RIM and Prolog has been achieved, making the fact that he is accessing an external database completely transparent to the user. Since the RIM database is therefore indistinguishable from the Prolog database, a virtual database is produced in which tuples stored in RIM and rules stored in Prolog are freely intermingled. This virtual database thus places all the power of Prolog at the RIM programmer's disposal (and conversely, gives all the power of RIM to the Prolog programmer). An immediate consequence of this result is that since it is possible to write rule-based systems in Prolog, it is possible to write them in the Prolog/RIM system YRIM; since it is possible to create "demons" in Prolog, it is possible to create them in YRIM, and since view definition, backtracking search of a problem space, and in fact the whole arsenal of AI techniques, are available in Prolog, they are available in YRIM as well. Furthermore, it is evident that our approach could have produced a similar integration of RIM and LISP or RIM and LProlog: in this case, RIM relations would have been available to the programmer as lists rather than predicates.

In view of the success and general applicability of this approach, what remains to be done? We have alluded above to the problems encountered when considering the application of YRIM to STRUTEX. Let us examine these issues in more detail, referring to the latest incarnation of STRUTEX available to us as an example. The main program can certainly be translated into YRIM with no difficulty:

```
strutex :- grfxec(0), dbexec(0),
           load, surface, support,
           grfxec(9), dbxec(9).
```

grfxec and dbxec would be defined in terms of <u>escape</u> predicates, e.g.:

```
grfxec(N) :- escape([N],Ierr).
```

New cases to handle the new action codes engendered by grfxec and dbxec would have to be added to RIM_MOD. Such an extension is straightforward, but represents only the tip of the iceberg. Consider routines LOAD, SURFACE, and SUPPORT. While these routines do contain interactions that have a classical expert systems flavor, major portions are devoted to double-precision computation, storage of temporaries and results in COMMON, and manipulation of the RIM database. Manipulation of RIM from Prolog is straightforward, of course: the <u>escape</u> predicates described in this report do exactly that. But these predicates are generally concerned with passing control information to RIM that prompts RIM to transmit data to Prolog, which is then manipulated by Prolog, and may be passed back to RIM as updates. This is where the problem arises: the manipulations must be

operations which the calling language (Prolog in YRIM) can perform easily. Prolog was able to perform the operations involved in TURBO, which were almost entirely manipulations of symbolic data. STRUTEX requires extensive operations on numbers, and double-precision floating point numbers at that. This, to put it mildly, is not Prolog's forte: many Prolog implementations (including York Prolog) do not feature floating-point numbers at all.

We have previously mentioned a solution to this problem: escape codes can be created to perform such operations; calls such as

escape([floatadd,'2.337','-88.6'],X)

can then perform the required manipulations. But this expedient is extremely cumbersome. Note that the "numeric" arguments in this call are actually strings; string/number and number/string conversions must therefore take place on both sides of the interface.

The reader will have spotted a more serious difficulty, however: uncontrolled proliferation of escape codes. If many of the operations that Pascal and FORTRAN do easily are to be performed on data that has been passed to Prolog, then an ever-growing set of escape codes will be required to pass this data back to the other side of the interface with appropriate instructions for action to be taken. The results of these actions will then need to be converted into a form Prolog can accept and passed back to Prolog, which presumably will soon

pass them back yet again for more manipulation.

All this buck-passing has the flavor of a badly organized bureaucracy. The problem arises from the fact that an AI language is being required to perform non-symbolic operations for which it is not designed. The appropriate organization for STRUTEX is the opposite of the present system's: rather than an AI language calling on a Pascal/FORTRAN/RIM system for non-symbolic conputations whose arguments and results it is ill-equipped to handle, a Pascal or FORTRAN program should be in control, and should call the AI system as subroutine whenever symbolic computations are required.

It should be noted that this requirement for reorganization is a result not of design defects in YRIM, but rather of the balance between symbolic and non-symbolic computation that prevails in a given application. The organization of YRIM is perfect for a largely symbolic expert system application such as TURBO, but inappropriate for a computation-intensive program such as STRUTEX. We have advocated for some time the concept of expert rules being callable as subroutines as needed by non-symbolic programs, and have in fact developed elsewhere implementations of some of these ideas.

What is proposed for future research, then, is an investigation of design concepts appropriate for integrating experise into largely non-symbolic programs such as STRUTEX. One possible approach has already been described in our discussion of a transition-diagram-based design for STRUTEX. A number of

related techniques appear promising, including investigations into the representation and storage (in RIM) of code sequences, generalizing the action codes currently transmitted by _escape_ predicates. While it would be premature to commit ourselves to any detailed proposal here, it is definitely envisioned that the resulting system will no longer be dependent on Prolog, and will be easily integrable into applications such as STRUTEX.

REFERENCES

Abbott, K., Exploration of Expert Systems Concepts for Onboard Diagnosis of Faults in a Turbofan Aircraft Engine, 1985 American Control Conference, June 1985, Boston, MA.

Feyock, S., Syntax Programming, Proceedings of the AAAI 1984 National Conference, August, Austin, Texas

Foderaro, F., Franz Lisp User's Guide, IRI, California State University, Los Angeles, CA, November 1981.

Gries, D., Compiler Construction for Digital Computers, J. Wiley & Sons, 1971.

Spivey, J., Portable Prolog User's Guide, Dept. of Computer Science, University of York, Heslington, York, England, October 1983.

BCS RIM Version 6 User Guide, Boeing Commercial Aircraft Company: Central Scientific Computing Complex Document Z-3, Nasa/Langley Research Center, May 1985