PUC

Series : Monografias em Ciência da Computação

No. 5 /84

EXPERT HELPERS TO DATA-BASED INFORMATION SYSTEMS

A. L. Furtado and C. M. O. Moura

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

Series : Monografias em Ciência da Computacao

No. 5 /84

Series Editor: Antonio Luz Furtado August 1984

EXPERT HELPERS TO DATA-BASED INFORMATION SYSTEMS*

A. L. Furtado, C. M. O. Moura +

* Research partly sponsored by FINEP and CNPq

+ Latin American Systems Research Institute, Rio de Janeiro, IBM do Brasil

ABSTRACT

This paper discusses some features of software tools - expert helpers - to provide expert assistance in the specification, usage and maintenance of data-based information systems. A simple PROLOG prototype and an example information system are used to illustrate the discussion.

KEYWORDS

data bases, formal specification, expert systems.

RESUMO

Este trabalho discute alguns aspectos de ferramentas de programação - ajudantes especialistas - para auxiliar na especificação, uso e manutenção de sistemas de informação baseados em bancos de dados. Um protótipo simples em PROLOG e um sistema de informação exemplo são utilizados para ilustrar a discussão.

PALAVRAS CHAVE

bancos de dados, especificação formal, sistemas especialistas.

1. Introduction

Expert systems have achieved considerable success in coping with specialized application areas [St]. They usually include a knowledge base, i.e. a repertoire of general rules, besides a conventional data base consisting of given facts. Through an inference capability, new facts are derived as combined consequences of the general rules and the given facts.

The present research considers some desirable features of software tools, to be called Expert Helpers (EHs), to aid in the specification, usage and maintenance of information systems. EHs combine the purpose of other software tools. Like data dictionaries they document the information system. Like running specifications, they allow experimenting with the information system. Their additional distinguishing feature, which justifies calling them "expert", is the inferential capability which, in special, can use general rules to synthesize sequences of function applications leading from the current state to a state where certain desired facts hold or cease to hold (see [SMF] and, for plan-generation in general, [Ni]).

To make the discussion more concrete, we shall refer to a simple prototype EH that is being developed modularly in micro-PROLOG [CEM]. The prototype contains two kinds of modules: specific and general. The specific modules refer to an application, i.e. a particular data-based information system. In the application module the conceptual schema of the application is specified, while view modules cover external schemas. The general modules permit, given an application, selectively interrogate its conceptual schema (the dictionary module) or run experiments on example data, both at the conceptual and at the external schema level (the processing module); another general module provides an interface with the operational data base proper, via shared sample files.

2. The specific modules

2.1. Application

Application modules contain the specification of some application at the conceptual schema level. Taking the abstract data type approach, we regard an application as an abstract data type, which can therefore be specified in terms of the functions defined on it [BZ].

The query functions correspond to the (conceptual) facts. The update functions are defined by:

- the domains from which come the values of their parameters;
- each fact added by each function;
- each fact deleted by each function,
- the (pre-) conditions to apply each function.

The effects of functions are the facts they add and/or delete.

Besides the additions and deletions originally intended, others may be required to preserve the static and transition integrity constraints [NY]. These may also require (pre-) conditions to the application of certain functions. Such conditions consist mainly of sets of positive and/or negative facts that must hold at the state immediately before the application of a function; conditions may be complemented by a logical expression.

Clearly this function-oriented style of specification defines the (update) functions explicitly, whereas facts are only implicitly mentioned as participating in the conditions and effects of functions.

As an overly simplified example, we consider an academic data base where the facts are that courses are offered and that students take courses. The integrity constraints require that students can only take offered courses (static constraint) and that the number of courses taken by a student cannot drop to zero during the academic term (transition constraint). The functions are:

```
- offer (x), with x∈course
  facts added: {offered(x)}
  facts deleted: { }
  conditions: { }

  cancel(x), with x∈course
  facts added: { }
  facts deleted: {offered(x)}
  conditions: {¬takes(z,x), for any z∈student}

- enroll(x,y), with x∈student, y∈course
  facts added: {takes(x,y)}
  facts deleted: { }
  conditions: {offered(y)}

- transfer(x,y,z), with x∈student, y∈course, z∈course
  facts added: {takes(x,z)}
  facts deleted: {takes(x,y)}
  conditions: {offered(z)}
```

We claim that this kind of specification is relatively simple and does not require that designers have a profound theoretical background. The only somewhat more involved problem is that of deriving the conditions (and additional effects, a case that does not occur in our example) necessary to enforce the constraints. Such problem, of course, must be attacked in a systematic way, as argued in [VF].

And yet, although the specification seems to say all that we need to know about the application, it is still incomplete. We should say what happens

(a) if a condition fails;
(b) if a fact that should be added is already present or a fact

that should be deleted is absent;

as also

(c) what happens with facts not explicitly mentioned as added or deleted by a function (this corresponds to the well-known frame problem).

Most specification methodologies either answer questions like these by introducing assumptions informally or by supplying additional rules. The former solution lacks rigour. The latter tends to clutter the specification with rules that look extraneous to the application and whose number may grow combinatorially. We prefer to provide a formal general solution to questions (a), (b), (c) that seems acceptable for a broad class of applications and is specified once and for all (at section 3.2).

2.2. Views

The view modules contain the specification of external schemas. They consist, first of all, of a number of external facts, implicitly defined by their mappings onto conceptual facts. We must stress that there may be more than one conceptual state for the same external state.

The module may also define the predicate, is-a, as developed in connection with semantic hierarchies in the area of artificial intelligence. With this predicate we can introduce new domains through a view and declare that they are specializations of given conceptual domains. Objects of the specialized domains inherit the properties of the corresponding more general ones, besides having new properties of their own.

We have included one view for our example application (the view of a user i, or the ith-view). The ith-view introduces the domain:

- lab, where lab is-a course

and the external facts:

```
- beginning(x), with x∈student
    mapping: {takes(x,y), with y∈(course - lab)}
             condition: { }

- practicing(x), with x∈student
    mapping: {takes(x,c2) ^ takes(x,y), with y∈lab}
             condition: {takes(x,c2) at the current state}
```

The specification requires that, to acquire the practicing status, a student must already be taking at the current state (and should continue to take) the co-requisite course c2.

View modules may also contain the definition of external update functions. For simplicity we preferred not to have that, assuming that views would be updated through sequences of conceptual update functions; as seen, the mappings of external facts may impose special conditions to the application of such sequences.

3. The general modules

3.1. Dictionary

The dictionary module allows to ask questions about the specification. Dictionaries are often defined as "meta data bases", in that they contain facts about what kinds of facts (and other elements) are maintained in the data base. At the present stage of our prototype, the dictionary module informs only about application modules. Using it we may ask:

- with respect to facts:
  . what kinds of facts are maintained;
  . from what domains come the values of their parameters;
  . what functions can add their occurrences;
  . what functions can delete their occurrences;

- with respect to functions:
  . what functions are defined;
  . from what domains come the values of their parameters;
  . what are the conditions for their application;
  . what kinds of facts they can add;
  . what kinds of facts they can delete.

Since only functions are explicitly defined, the questions on the facts are answered by looking at how facts are referred to in the function specifications. This gives us the possibility to check if such references have been done consistently. For example, we can adapt the question

- from what domains come the values of the parameters of a fact

so that we can verify whether all functions involving the fact have parameters in the same corresponding domains. Suppose we had in our example application a function f(x,y) with the effect of deleting takes(x,y), and suppose further that the parameters of f were declared x∈course, y∈course. Clearly this would be inconsistent with the other functions, which expect x to be a student in takes(x,y).

3.2. Processing

The processing module allows to run the specification, as expressed declaratively in specific application and view modules. It also fulfills the fundamental role of completing such specifications by providing solutions to questions (a), (b), (c) of section 2.1.

First, we introduce the concepts of:

- valid application of a function - the application of a function is valid if the function has been invoked with the correct number of parameters of the prescribed domains (syntactical validity), and all conditions for its application are preliminarly satisfied (semantic validity);

- productive application of a function - the application of a function is productive if all facts that the function should delete are present and all facts that it should add are absent before its application.

Now we can say that an assertion holds in a data base state only if one of the cases below applies:

. case 1 - the assertion corresponds to a stored fact, which means that it holds at the current state;
. case 2 - the assertion corresponds to a fact to be added by a function, and therefore holds at the state reached by a valid and productive application of the function;
. case 3 - the assertion corresponds to a fact that is not deleted by a function; so, if it held at the state whereto the function is applied, it also holds at the state reached by its application.

Another predicate, not-holds, is defined dually to handle negative assertions. The definition of holds and not-holds is complemented by the criterion known as negation as failure [Cl], according to which an assertion is false unless provable from the, possibly recursive, stated rules.

We conclude that either a function meets all its syntactical and semantic requirements and achieves all its effects and only these, or it does nothing at all and hence no state change is operated. This provides an answer to questions (a), (b), (c), which is precisely stated through the axioms of the processing module (plus negation as failure).

To write the axioms we have followed the idea, introduced in [Ko], to treat facts and functions as terms. Other axioms adapt an algorithm [Wa] to derive alternative sequences of function applications leading to a state (partially) characterized by an indicated set of positive and/or negative facts. Such set is treated as a conjunction of possibly interacting goals; interaction induces a partial order on function application. Conditions to function application are established as sub-goals. If there is an inconsistency among goals or sub-goals, no sequence is generated.

The sequences of function applications are represented internally in a nested notation but are displayed in a more readable "linear" format. To compare alternative sequences, we can examine what facts each sequence adds, deletes or preserves. We can actually execute a chosen sequence to obtain a new current state.

Finally, the module also allows views to be queried or updated, by working on the underlying conceptual facts and states. To determine the generation of external states, however, the present version only permits the indication of sets of positive external facts.

3.3. Interface

The interface module permits a simple communication between the EH and the associated data base via shared sample files, i.e. subsets of the operational files whose contents are considered representative of the situations that can arise.

Two features are available, allowing to

- read a sample file and add its contents as new facts of a given kind;

- use the facts of a given kind, holding at the current state, to write a sample file.

Since we are running micro-PROLOG under CP/M, sample files are standard format CP/M files. The relational DBMS dBASE II [AT] is being used in our experiments. We had to make provision for little problems, such as expecting or not an end-of-file mark, filling or not trailing positions with blanks, updating the micro-PROLOG dictionary, etc. The sample CP/M files can be written or read by dBASE II, using the commands "copy to <file> for <expression> sdf" and "append from <file> sdf", respectively.

All modules are fully listed in the APPENDIX. We must warn the reader that different releases of any system tend to require minor adjustments (our release of micro-PROLOG is 3.0). The syntax follows the general format of PROLOG versions, especially notable deviations being that binary predicates are displayed in infix notation and "if" is substituted for the more usual left-directed arrow. It provides a set-former capability [Wa2] that we have used extensively. Our immediate plans include a thorough revision to improve efficiency; in particular, considerable time is being taken for the processing module to convince itself that it cannot generate any further sequences.

4. Using the prototype

We begin showing how to learn about the conceptual schema of an application.

-- Dictionary look-up:
   (load the application and the dictionary modules)

- "what kinds of facts are kept ?"
```
  which(x facts(x))
  answer: ((takes X Y) (offered Z))
```

```
- "what are the domains of takes ?"
  which(x fact-domains(takes x))
  answer: (student course)

- "which functions add takes ?"
  which(x fact-adders(takes x))
  answer: ((transfer X Y Z) (enroll x y))
```

To conduct experiments with example elements, we introduce:

```
instance(student John)
instance(student Peter)
instance(course c1)
instance(course c2)
```

and a current state, s0, with the following facts (noting however that we have always the option to start with the "empty" state, consisting only of the dummy fact()):

```
fact((offered c1))
fact((offered c2))
fact((takes John c1))
fact((takes Peter c2))
```

-- Querying the current state
(load the application module)

```
- "does John take c1 ?" (a yes/no-question)
  does(fact((takes John c1)))
  answer: YES

  "what courses are offered ?" (a wh-question)
  which(x fact((offered x)))
  answer: c1
          c2
```

-- deriving sequences of function applications
(load the application and the processing modules)

```
- "how can we reach a state where c1 is not offered and John
  takes c2 ?" (a how-to-do-question)
  one(x plans(((not offered c1) (takes John c2)) y) and
      linear(y x))
  answer: (s0 ; transfer John c1 c2 ; cancel c1)

- "what are the consequences of executing the sequence above ?"
  (a what-if-question)
  one((plus x minus y and remains z)
        plans(((not offered c1) (takes John c2)) X) and
        seq-added(x X) and
        seq-deleted(y X) and
        seq-preserved(z X))
  answer: (plus ((takes John c2)) minus ((takes John c1) (offered
          c1)) and remains ((takes Peter c2) (offered c2)))
```

--- Obtaining a new current state
(load the application and the processing module)

```
- "obtain a state where John takes c2."
  does(state(((takes John c2))))
  answer:  (s0 ; enroll John c2)
           (s0 ; transfer John c1 c2)
  choose one option
  .2
  YES
  inspecting the new current state
  list(fact)
  fact((offered c1))
  fact((offered c2))
  fact((takes Peter c2))
  fact((takes John c2))
```

--- Querying and updating the ith-view
(load the application, the view and the processing modules)

First we add a new element:

```
instance(lab c3)

- "what external facts hold corresponding to the conceptual state
  above ?"
  which(x external-fact(i x))
  answer:  (beginning John)
           (beginning Peter)

- "from  the previous state,  obtain a state where both John  and
  Peter are practicing."
  does(external-state(i ((practicing John) (practicing Peter)) ))
  answer:  (s0 ; offer c3 ; enroll John c3 ; enroll Peter c3)
           (s0 ; offer c3 ; enroll Peter c3 ; enroll John c3)
           choose one option
           .0 /* in this case no state-change is performed */
           NO
```

Communicating via sample files with a DBMS
(load the interface module)

```
  "read 'takes' facts from file alpha."
  does(in-predicate(takes "alpha.txt" ((CON 5)(CON 2)) (x y)))
  answer: YES
  /*  from this point on the added facts can be treated like  any
  others*

  "write all 'takes' facts on file beta."
  does(out-predicate(takes "beta.txt" ((CON 5)(CON 2))))
  answer: YES
   *  if the file does not exist it is created,  otherwise it  is
  overwriten */
```

5. Conclusions

It is a widespread conjecture that we shall see large intelligent data-based information systems in the near future. Our point in this paper is that, even if size and intelligence are incompatible with efficiency given the present state of the art, intelligence may be present in smaller software tools, "connected" somehow with the large systems, such as our proposed expert helpers.

The processing module of our EH can be regarded as an interpreter when it generates and executes a sequence of function applications. As seen, function applications may be conditional. Also, we have iteration, as may be called for by the conditions of the cancel(x) function: if there are students taking course x, the set {not takes(z,x)} will be established as sub-goal, causing the repeated invocation of transfer(z,x,y), for y∈course. So, the three main program control structures (sequence, selection and iteration) are present. Although we synthesize executions rather than programs [BF], a similar result is achieved, with the usual loss in terms of efficiency inherent in the choice of interpretive instead of compilation-oriented solutions.

The highly non-procedural ability of the prototype to perform conceptual and external state transitions, by indicating target sets of facts, is at present its most interesting feature. Other features, related to expert assistance at the specification phase and during the life-time of an application, should be identified by future research and gradually incorporated.

References

[AT] Ashton-Tate - "dBASE II assembly-language relational database management system" - reference manual - Ashton-Tate (1982).

[BF] A. Barr and E. A. Feigenbaum (eds.) - "Automatic Programming"- in "The handbook of artificial intelligence" - vol. 2, chapter 10 - HeurisTech Press and William Kaufman (1982) 295-379.

[BZ] M. L. Brodie and S. N. Zilles (eds.) - Proc. of the Workshop on Data Abstraction, Databases and Conceptual Modelling - SIGMOD Record, vol. 11, n. 2 (1981).

[CEM] K. L. Clark, J. R. Ennals and F. G. McCabe - "A micro-PROLOG primer" - Logic Programming Associates (1983).

[Ko] R. Kowalski - "Logic for problem solving" - North-Holland (1979).

[Ni] N. J. Nilsson - "Principles of artificial intelligence" - Springer-Verlag [1982].

[NY] J. M. Nicolas and K. Yazdanian - "Integrity checking in deductive data bases" - in "Logic and data bases" - H. Gallaire and J. Minker (eds.) - Plenum Press (1978) 325-344.

[SMF] C. S. dos Santos, T. S. E. Maibaum and A. L. Furtado - "Conceptual modelling of data base operations" - International Journal of Computer and Information Sciences - vol. 10, n. 5 (1981) 299 - 314.

[St] M. Stefik et al - "The organization of expert systems - a tutorial" - Artificial Intelligence - vol. 19, n. 2 (1982) 135-173.

[VF] P. A. S. Veloso and A. L. Furtado - "Towards simpler and yet complete formal specifications" - submitted for publication.

[Wa1] D. H. D. Warren - "WARPLAN: a system for generating plans" - memo 76 - University of Edinburgh (1974).

[Wa2] D. H. D. Warren - "Higher-order extensions to PROLOG: are they needed?" - in "Machine intelligence" - vol. 10 - J. E. Hayes, D. Michie and Y. H. Pao (eds.) - Halsted Press (1982) 441-454.

APPENDIX

A. Application module

```
(offer X) operation (course)
(cancel X) operation (course)
(enroll X Y) operation (student course)
(transfer X Y Z) operation (student course course)
(offered X) added (offer X)
(takes X Y) added (enroll X Y)
(takes X Y) added (transfer X Z Y)
(offered X) deleted (cancel X)
(takes X Y) deleted (transfer X Y Z)
conditions ((offer X) () Y)
conditions ((cancel X) Y Z) if
          Y Is-All ((not takes x X) (takes x X) holds Z)
conditions ((enroll X Y) ((offered Y)) Z)
conditions ((transfer X Y Z) ((offered Z)) x)
```

B. View module

```
map (i (beginning X) ((takes X Y)!Z)) if
        student domain X and
        () Is-All (x lab domain x and fact ((takes X x))) and
        course domain Y and
        Not (lab domain Y) and
        Z Is-All ((not takes X x) lab domain x)
map (i (practicing X) ((takes X c2) (takes X Y))) if
        student domain X and
        fact ((takes X c2)) and
        lab domain Y
lab is-a course
```

C. Dictionary module

```
facts (X) if
        Y Is-All (Z CL (((added Z x)|y) 1 z)) and
        Y no-duplicate X
X fact-domains Y if
        X fact-adders Z and
        (x|y) ON Z and
        x function-domains z and
        APPEND ((x) z X1) and
        CL (((added (X|Y) X1)|Y1) 1 Z1) and
        / ()
X fact-adders Y if
        Y Is-All (Z CL (((added (X|x) Z)|y) 1 z))
X fact-deleters Y if
        Y Is-All (Z CL (((deleted (X|x) Z)|y) 1 z))
functions (X) if
        X Is-All (Y CL (((operation Y Z)|x) 1 y))
X function-domains Y if
        CL (((operation (X|Z) Y)|x) 1 y)
X function-conditions Y if
        CL (((conditions (X|Z) x y)|z) 1 X1) and
        (x|z) simplify Y
X function-adds Y if
        Y Is-All (Z CL (((added Z (X|x))|y) 1 z))
X function-deletes Y if
        Y Is-All (Z CL (((deleted Z (X|x))|y) 1 z))
() no-duplicate ()
(X|Y) no-duplicate Z if
        X ON Y and
        / () and
        Y no-duplicate Z
(X|Y) no-duplicate (X|Z) if
        Y no-duplicate Z
(X) simplify X if
        / ()
X simplify X
```

D. Processing module

```
state (X) if
          Y Is-All (Z X plans Z and Z linear x and PP (x)) and
          y Is-All (z z ON Y) and
          PP (choose one option) and
          R (X1) and
          choose (y X1 Y1) and
          exec (Y1)
X plans Y if
          plan (X () s0 Y)
plan ((X|Y) Z x y) if
          / () and
          solve (X Z x z X1) and
          plan (Y z X1 y)
plan (() X Y Y) if
          / ()
solve ((not|X) Y Z ((not|X)|Y) Z) if
          X not-holds Z
solve ((not|X) Y Z ((not|X)|Y) x) if
          X deleted (y|z) and
          syntax-OK ((y|z)) and
          achieve ((not|X) (y|z) Y Z x)
solve ((not|X) Y Z x y) if
          / () and
          FAIL ()
solve (X Y Z (X|Y) Z) if
          X holds Z
solve (X Y Z (X|Y) x) if
          X added (y|z) and
          syntax-OK ((y|z)) and
          achieve (X (y|z) Y Z x)
achieve (X (Y|Z) x y (result (Y|Z) z)) if
          conditions ((Y|Z) X1 y) and
          plan (X1 x y z) and
          productive ((result (Y|Z) z)) and
          x preserved (result (Y|Z) z)
achieve (X Y Z (result (x|y) z) (result (x|y) X1)) if
          retrace (Z (x|y) Y1) and
          achieve (X Y Y1 z X1) and
          productive ((result (x|y) X1)) and
          conditions ((x|y) Z1 X1) and
          (X) preserved (result (x|y) X1) and
          Z1 preserved X1
retrace (((not|X)|Y) Z x) if
          X deleted Z and
          / () and
          retrace (Y Z x)
retrace ((X|Y) Z x) if
          X added Z and
          / () and
          retrace (Y Z x)
retrace ((X|Y) Z (X|x)) if
          retrace (Y Z x)
retrace (() X ())
```

```
X preserved Y if
          (Z preserved1 Y) For-All (Z Z ON X)
(not!X) preserved1 Y if
          / () and
          X not-holds Y
X preserved1 Y if
          X holds Y
syntax-OK ((X!Y)) if
          (X!Z) operation x and
          Y syntax-OK1 x
(X!Y) syntax-OK1 (Z!x) if
          Z domain X and
          Y syntax-OK1 x
() syntax-OK1 ()
X domain Y if
          X instance Y
X domain Y if
          CL (((is-a Z X)) 1 x) and
          Z domain Y
productive ((result (X!Y) Z)) if
          (x holds Z) For-All (x x deleted (X!Y)) and
          (Not (x holds Z)) For-All (x x added (X!Y))
X holds s0 if
          fact (X)
X holds (result (Y!Z) x) if
          X added (Y!Z) and
          syntax-OK ((Y!Z))
X holds (result (Y!Z) x) if
          ((VAR Y)) OR ((VAR Z)) and
          / () and
          FAIL ()
X holds (result (Y!Z) x) if
          / () and
          X holds x and
          Not (X deleted (Y!Z))
X not-holds s0 if
          Not (fact (X))
X not-holds (result (Y!Z) x) if
          X deleted (Y!Z) and
          syntax-OK ((Y!Z))
X not-holds (result (Y!Z) x) if
          ((VAR Y)) OR ((VAR Z)) and
          / () and
          FAIL ()
X not-holds (result (Y!Z) x) if
          / () and
          X not-holds x and
          Not (X added (Y!Z))
s0 linear (s0)
(result (X!Y) Z) linear x if
          Z linear y and
          APPEND (y (; X!Y) x)
```

```
choose (X 0 Y) if
          / () and
          FAIL ()
choose (((X|Y)|Z) 1 (X|Y)) if
          / ()
choose (((X|Y)|Z) x y) if
          SUM (1 z x) and
          choose (Z z y)
exec (s0)
exec (X) if
          Y seq-added X and
          Z seq-deleted X and
          (Add ((fact (x)))) For-All (x x ON Y) and
          (Delete ((fact (y)))) For-All (y y ON Z)
X seq-added Y if
          X Is-All (Z Z holds Y and Not (Z holds s0))
X seq-deleted Y if
          X Is-All (Z Z holds s0 and Not (Z holds Y))
X seq-preserved Y if
          X Is-All (Z Z holds s0 and Z holds Y)
X external-fact Y if
          map (X Y Z) and
          external-fact1 (Z)
external-fact1 (())
external-fact1 ((X|Y)) if
          fact (X) and
          external-fact1 (Y)
external-fact1 (((not|X)|Y)) if
          external-fact1 (Y) and
          Not (fact (X))
maps (X () ())
maps (X (Y|Z) x) if
          map (X Y y) and
          maps (X Z z) and
          APPEND (y z x)
X external-state Y if
          Z Is-All (x maps (X Y x)) and
          Z states y and
          states1 (y)
() states ()
(X|Y) states Z if
          x Is-All (y X plans y and y linear z and PP (z)) and
          X1 Is-All (Y1 Y1 ON x) and
          Y states Z1 and
          APPEND (X1 Z1 Z)
states1 (X) if
          PP (choose one option) and
          R (Y) and
          choose (X Y Z) and
          exec (Z)
```

E. Interface module

```
in-predicate (X Y Z x) if
          OPEN (Y) and
          Y SEEK (0|0) and
          APPEND (Z ((CON 2)) y) and
          in-predicate1 (X Y y x)
in-predicate (X Y Z x) if
          CLOSE (Y) and
          clean-dict ()
in-predicate1 (X Y Z x) if
          x copy y and
          APPEND (y (z) X1) and
          FREAD (Y Z X1) and
          y no-blank-list Y1 and
          Add ((fact ((X|Y1)))) and
          in-predicate1 (X Y Z x)
out-predicate (X Y Z) if
          CREATE (Y) and
          Y SEEK (0|0) and
          APPEND (Z ((CON 2)) x) and
          eor (y) and
          (APPEND (z (y) X1) and FWRITE (Y x X1)) For-All (z fact ((X|z))) and
          CLOSE (Y)
() copy ()
(X|Y) copy (Z|x) if
          VAR (X) and
          Y copy x
(X|Y) copy (X|Z) if
          Not (VAR (X)) and
          Y copy Z
eor (X) if
          Y CHAROF 13 and
          Z CHAROF 10 and
          (Y Z) STRINGOF X
clean-dict () if
          (Delete ((dict (fact)))) For-All (X dict (fact)) and
          Add ((dict (fact))) and
          Delete ((dict (dict)))
X no-blank Y if
          Z STRINGOF X and
          x Is-All (y y ON Z and Not (y CHAROF 32)) and
          z Is-All (X1 X1 ON x) and
          z STRINGOF Y
X no-blank-list Y if
          Z Is-All (x y ON X and y no-blank x) and
          Y Is-All (z z ON Z)
```