EÖTVÖS LORÁND UNIVERSITY

FIRST SEMINAR ON ARTIFICIAL INTELLIGENCE

January 23-24, 1989

Visegrád, Hungary

Budapest, 1989

EÖTVÖS LORÁND UNIVERSITY

FIRST SEMINAR ON ARTIFICIAL INTELLIGENCE

January 23-24, 1989
Visegrád, Hungary

Edited by I. Fekete and S. Nagy
Eötvös Loránd University
Department of General Computer Science
H-1117 Budapest, Bogdánfy u. 10/B

Budapest, 1989

Készült az ELTE Sokszorosítóüzemében
200 példányban
Felelős kiadó: Dr. Klinghammer István
Felelős vezető: Arató Tamás
ELTE 89349

FOREWORD

This book contains papers presented at the First Seminar on Artificial Intelligence held 23-24. January, 1989, in Visegrád, Hungary. This workshop was organized by the Department of General Computer Science of Eötvös Loránd University, on the initiative of Prof. L. Varga, the head of the department.

The importance and impact of artificial intelligence and expert system applications shows a growing tendency in our country, too. Even more efforts are made in searching theoretical issues, as well as in developing systems and tools for applications, and in elaborating and teaching educational material for students. The research groups, however, seem to be working a bit isolated. Thus the aim of the seminar was to give an opportunity to scientists to meet, to get information about each other's results, to exchange ideas, and to harmonize conflicting opinions.

It is hoped that the range of the contributions covers a broad spectrum of artificial intelligence research and gives a representative image of the results in Hungary. The proceedings presented here and the lectures delivered at the seminar are not completely the same, but the difference is not significant.

Special thanks are due to all lecturers, to the authors of the articles, and to all participants of this scientific meeting.

The Second Seminar on Artificial Intelligence is planed to take place at the beginning of 1991.

Budapest, January, 1989.

The Editors

CONTENTS

FAIR, A FRAME-BASED SYSTEM INTEGRATED WITH MPROLOG

P. Ecsedi-Tóth, A. Péter Wágner

Computer Research and Innovation Centre
Budapest, 1Ø15, Donáti u. 35-45.

1. INTRODUCTION

1.1 FRAME-BASED REPRESENTATION OF KNOWLEDGE

One of the most significant trends in modern expert system shell developments is the useage of several knowledge representation methods and programming paradigms [4],[2]. In the resulting hybrid shells one may use, for example, some or all of the following representation techniques: logic-based, rule-based, frame-based, semantic nets, isomorphic (or direct) and procedural representations. Two of these methods, namely, the rule-based (or more generally, logic-based) and the frame-based ones seem to play an outstanding role in representation tasks. We think that in a modern expert system shell at least these two techniques must be available. On the other hand, they seem to be sufficient, too: every other (known) knowledge representation techniques can easily be obtained from these two ones. This belief is supported by statistical data, too. For example, in [5] a statistical evaluation is published on the Japanese expert system projects. According to these statistics, more than 8Ø-9Ø per cent of the expert systems were developed by using these two basic representational forms. Rule-based representation is relatively well-understood and can be used with ease. Nevertheless, some problems arise if the number of rules is large (above 1ØØØ), since the knowledge base containing the rules is ill-structured. The frame-based approaches, on the other hand, provide useful possibilities to structure the knowledge appropriately. Indeed experiences show that building a system using frames reduces the required time to two third of the time needed to build the same system by rules. What is more, the measure of relative reduction is increasing if the size of the system is

increasing. Frames are, however, basically static objects, and it is not so easy to compile dynamic phenomena into frames. So ideally, the two approaches, i.e. frames and rules should be used in an integrated way. In fact we would like to describe here a particular system FAIR (Frames in AI Representation of knowledge), which allows the user to merge frames and rules in a flexible manner.

1.2 FRAMES AND MPROLOG

If the knowledge is represented formally somehow, we want to manipulate it. Most of the previously mentioned knowledge representation methods have a dedicated paradigm which can be used to put them into use. For example, logic-based and rule-based representations are usually used by the so called "logic programming" paradigm. A typical paradigm by which frame-based knowledge can be put into work is known as "object-oriented" programming. In object oriented programming one looks at a frame (or "object") as a unit of some concepts and the algorithms which manipulate these concepts, and uses a loose (called "blackboard") or close (called "message passing") synchronization among the units. Thus, when we want to integrate frames and rules, first we have to find such programming paradigms which are appropriate for both representations. Fortunately, a closer look on the problem indicates that the connection between formal representation methods and their "dedicated" paradigms are not so tight as they looked at first glance [1]. Indeed, our main aim here is to show that frames (and of course rules) can be used together with the logic programming paradigm.

In order to be specific, in our practical work we concretize these notions somewhat: by rule-based representation of knowledge and its dedicated manipulation paradigm "logic programming" we simply mean the use of PROLOG programming (in particular, we use MPROLOG).

By incorporating frames into PROLOG we obtain two "orthogonal" sublanguages which can be used in an arbitraryly merged manner: If pure PROLOG is needed, then we can use it without any inconvenience, and , on the other hand, if we wish to use frames only, then we can do it without difficulties; moreover, we are completely free to choose the point between the two extreme cases where our particular problem will be solved.

We may start, for example, at the outermost level with an MPROLOG program which uses some frames, then we can define these frames

which may contain again MPROLOG programs and so on. Of course we may start from some frames and go inside by defining logic programs (rules), too. Thus the proposed integrated language is very much more expressive then a simple two-level language composed e.g. from frames (resp. rules) in the outer level and from logic programs (resp. frames) in the inner one because the two sublanguages can be embedded into each other in an arbitrary depth.

1.3. THE FAIR

The heart of the FAIR system is an integrated representation language. This language makes it possible to use several knowledge representation methods such as frames, semantic nets, scripts, graphs, and logic formulae, production rules, and procedural information (the latter three in MPROLOG). The integrated language is supported by a dedicated user-friendly editor, or can be used in line mode or with menus.

From the "programming paradigm" point of view the FAIR supports only the logic programming. Nevertheless, the language of FAIR is very easily extendible by other means among them by new manipulation methods (i.e. by new paradigms). This flexibility and extendibility are considered to be among the main advantages of the FAIR system.

In the next section we shall describe the frame-based sublanguage of FAIR. The other sublanguage MPROLOG is assumed to be known (in fact, familiarity with general properties of PROLOG-like languages, see e.g.[3], is enough for understanding this paper; special properties of MPROLOG are used here only in a transparent way).

2. THE FRAME LANGUAGE OF FAIR

Information is treated in FAIR on three different levels:

- the level of frames
- the level of frame structures
- the level of worlds.

In this section we shell discuss these levels in some details.

2.1. SIMPLE FRAMES

A frame is a structured abstract model of a concept. This model is usually given by an unordered set of properties. These properties are called slots; each slot represents an aspect of the concept considered important from the point of view of the description. A slot may have values; these values can be names of other frames or

expressions of some formal/computer language (in our particular case of MPROLOG).
We note that some parts of a frame can be omitted as is indicated by the following general syntax:

```
frame: name;
   [slot_1:[[value_11,value_12,...,value_1n1]];]
   [slot_2:[[value_21,value_22,...,value_2n2]];]
   ..........................................
   [slot_m:[[value_m1,value_m2,...,value_mnm]];]
end
```

Observe that the outermost and innermost brackets denote simply the optional parts of the definition, while the brackets inbetween (in boldface) are keywords and denote lists (in MPROLOG).
If a frame has only a name (and no slots or values are present) then we call it "primitive frame". If a frame has a name and some slots but no values, then it is called a "generic frame". Finally, if nothing is omitted, then we say it to be a "full frame".
Below we shall give some examples which illustrate these points.

```
frame: polygon;
end

frame: rhomb;
  base:[];
  alpha:[];
end
```

```
frame: rhomb_2;
  base:[40];
  alpha:[30];
  similar_to:[rhomb_1];
end
```

Operationally frames are simply stored in a retrievable form in the knowledge base. These manipulations can be carried out by using some operations provided by the FAIR system. For example, the frame called "rhomb" can be stored in the following manner:

```
create_frame(rhomb,base).
create_frame(rhomb,alpha).
```

(NOTICE. For the human user the FAIR provides a menu-based dedicated editor for entering frames in a much more convenient way, than described above. The above form has the advantage that it can be used from programs, too.)
Assuming that the frame "rhomb_2" is stored in the form given above, we may execute the predicate:

```
access_value(rhomb_2,base,X).
```

As a result the value [40] will be returned in the variable X.

Among others, frames can be used in MPROLOG rules as abstract data types. This kind of usage is supported by the predicate "**strieve**". Actually, "**strieve**" is a combined "store and retrieve" (or more precisely, "create and access"). The bound values given as parameters of the strieve predicate will be stored in the frame while the system will try to determine by inheritance, demon call or else (see later) the values of all unbound variables in the argument of the strieve. For example, consider the predicate

```
strieve(parallelogram(base(B),side(S))).
```

The result of this predicate depends on whether B and S have a value or not. The extreme cases are as follows:

- B, S have fixed values. Then these values will be stored in the frame parallelogram, in the slots "base" and "side".
- B, S have no values. Then the system will try to access the slots "base" and "side" in the frame "parallelogram".

2.2. META-INFORMATION

In order to cope with the intrinsic complexity of real world concepts, additional information (called here "meta-information") can be associated to the main parts of frames. Thus, we may associate meta-information to frames themselves, to the slots or to values. Accordingly, meta-information can be separated from the main body of the original frame by the following keywords

```
meta_frame          meta_end
meta_slot           meta_end
meta_value          meta_end
```

The meta-information will also be coded by frames. Basically, there are two ways to define frames holding meta-information: firstly the user can give his/her own frame for this purpose and secondly the user can fill-in a pre-fabricated frame tailored specially for holding meta-information. These pre-fabricated frames may contain information about

- demons or other activities concerning frames and slots
- constraints on the cardinality and range of the values of a slot
- where a missing value can be obtained from; possibilities include default values, inheritance from other frames or asking prescribed questions from the user.

For the sake of illustration, consider the following frame:

```
frame: polygon;
   number_of_edges:[];
      meta_slot: auxiliary_frame;
         range:[3,4,5,6,7,8,9];
         default:3;
      meta_end
end
```

If a value will be provided to the slot "number_of_edges" then the system will check whether or not the offered value is in the given range. For example, if we try to add 1 or 2 then the system will reject the trial; on the other hand, 3 or 4 or more up to 9 will be accepted.

An outstanding role is played by the demons associated to frames or slots. In our geometric example, we shall also use some demons which are sensitive to the access of a value of a particular slot. These are called of type "if_accessed_demon". For illustration, see the following frames and MPROLOG program:

```
frame: equiangle;
   alpha:[];
      meta_slot:alpha_meta;
         if_accessed_demon:[alpha_demon];
      meta_end
end

frame: alpha_demon;
   is_a:demon;
   activity:[alpha_act];
end

alpha_act:-
         access_value(work,frame_name,[FN]),
         access_value(FN,number_of_edges,[N]),
         X is 180-360 div N,
         create_frame(equiangle,alpha,X,overwrite).
```

We see that the slot "alpha" in the frame "equiangle" has an associated "if_accessed_demon" with name "alpha_demon". If a trial is made to access the slot "alpha" in this frame, the demon will be activated (after the trial, but before the value (if exists) is returned; this is the default way of invocation of an "if_accessed_demon" which can be modified if necessary). The demon itself is defined in two steps; first we filled-in the pre-fabricated standard frame devoted to defining demons and second, we defined the activity to be carried out by the demon as an MPROLOG program. According to these definitions, the demon, when activated,

will read out the value of the slot "frame_name" from the frame "work" and then it reads out the value N of the slot "number_of_edges" from the frame with name just retrieved from "work"; then the value of E will be computed by dividing 360 by N and retracting the result from 180; finally the value of E will overwrite the old value (if any) of the slot "alpha" in the frame "equiangle".
Physically we may give meta-information in two ways:
Firstly, we may use the editor of FAIR and write frames as in the example above. Then the system will automatically associate the meta-information in "alpha_demon" to the slot "alpha" in "equiangle".
Secondly, we may define the frame holding the meta-information separately from the frame to which it is to be associated and then we may use the **associate** predicate in line-mode or from a menu in order to manipulate the things. For example, we may define the frames

```
frame: equiangle;
  alpha:[];
end
```

```
frame: alpha_demon;
  is_a:demon;
  activity:[alpha_act];
end
```

and then we may use the predicate

associate(equiangle,alpha,alpha_demon).

to connect the two frames. Disconnection of the meta-information is carried out in each case by the predicate **dissociate**. For example, we may use

dissociate(equiangle,alpha,alpha_demon).

in order to destroy the connection between the frames "equiangle" and "alpha_demon".

2.3. RELATIONS

As concepts modelled by frames can be connected somehow, we may want to define relations among frames. In FAIR there exists two kinds of relations:

- Relations known to the system by default
- Relations defined by the user.

Once a relation has been defined, its name can be used as a slot in the frame to be connected to another, while the value of this slot identifies the frame to which we want to connect our particular one. The FAIR system knows about two relations, namely the "is_a" and the "instance_of" relations. Actually, these two relations are treated by FAIR in the same way. Intuitively, however, we may use "is_a" as the familiar "subset" relation and "instance_of" as the "membership" relation of set theory.
User can define also relations by filling-in a pre-fabricated generic frame called "standard_relation":

```
frame: standard_relation;
   domain:[];
   range:[];
   inverse:[];
   demon:[];
   inheritance:[];
end
```

The slots "**domain**" and "**range**" define the first and second argument of the relation. (Observe, that FAIR supports only binary relations. Unary relations and relations with more than two arguments can be implemented by binary relations easily.) In the "**inverse**" slot we may specify the name of the inverse relation (the inverse is meant to be the total inverse; this will be defined automatically together with the original relation). Demons can be associated to relations, too, in the slot "**demon**". Finally, the value of the slot "**inheritance**" specifies the information which is allowed to pass along the relation. We shall discuss the possibility of controlling inheritance later.
If we fill-in this pre-fabricated frame we must add the slot "is_a" with the value "[relation]". To illustrate the point we define the relation "similar_to" needed in the geometry example:

```
frame: similar_to;
   is_a:[relation];
   inverse:[similar_to];
   inheritance:[similarity_inheritance];
end
```

Notice that the slots "**domain**" and "**range**" are omitted here: the FAIR system will apply the default values "all", i.e. the relation "**similar_to**" can be used between arbitrary frames in the knowledge base. In the case of "**demon**" slot, on the other hand, the system

will do nothing since this is the default. The inheritance specification will be explained below.

2.4. INHERITANCE

Relations defined among frames allow transfer of information from one frame to another. This mechanism makes it possible to store every piece of information at the most appropriate place.
In the FAIR system, inheritance may be controlled in two different ways:

- Local control can modify the inheritance of a slot by specifying the way how the slot may inherit information from other places and the way how the slot can be inherited by other places
- Global control can modify the inheritance along a particular relation

The information needed to control inheritance will be given in pre-fabricated frames.
The frame specifying local control can be filled-in by the user; if it is filled in, its name must be given in the meta-information associated to the slot (the inheritance of which is to be controlled). Actually, we may forbid the inheritance of the (value of the) slot or may determine the strategy of search for the next candidate (from which the lacking information will be tried to inherit) and finally we may give "hints" for the inheritance mechanism.
The global control of inheritance affects the information which may pass along a relation. The necessary information is given in the following frames:

```
frame: standard_relation_inheritance;
  excluded_slots:[];
  ex_condition:[];
  included_slots:[];
  in_condition:[];
end
```

The user may fill in this frame; then its name can be given as the value of the "**inheritance**" slot in the defining frame of the relation to be controlled.
The value of the slot "**excluded_slots**" (resp. "**included_slots**") can be a list of slot names. The specified slots will be excluded (included) when inheritance takes place if the appropriate conditions given in the "**condition**" slot hold, otherwise the

Once a relation has been defined, its name can be used as a slot in the frame to be connected to another, while the value of this slot identifies the frame to which we want to connect our particular one. The FAIR system knows about two relations, namely the "is_a" and the "instance_of" relations. Actually, these two relations are treated by FAIR in the same way. Intuitively, however, we may use "is_a" as the familiar "subset" relation and "instance_of" as the "membership" relation of set theory.
User can define also relations by filling-in a pre-fabricated generic frame called "standard_relation":

```
frame: standard_relation;
   domain:[];
   range:[];
   inverse:[];
   demon:[];
   inheritance:[];
end
```

The slots "**domain**" and "**range**" define the first and second argument of the relation. (Observe, that FAIR supports only binary relations. Unary relations and relations with more than two arguments can be implemented by binary relations easily.) In the "**inverse**" slot we may specify the name of the inverse relation (the inverse is meant to be the total inverse; this will be defined automatically together with the original relation). Demons can be associated to relations, too, in the slot "**demon**". Finally, the value of the slot "**inheritance**" specifies the information which is allowed to pass along the relation. We shall discuss the possibility of controlling inheritance later.
If we fill-in this pre-fabricated frame we must add the slot "is_a" with the value "[relation]". To illustrate the point we define the relation "similar_to" needed in the geometry example:

```
frame: similar_to;
   is_a:[relation];
   inverse:[similar_to];
   inheritance:[similarity_inheritance];
end
```

Notice that the slots "**domain**" and "**range**" are omitted here: the FAIR system will apply the default values "all", i.e. the relation "**similar_to**" can be used between arbitrary frames in the knowledge base. In the case of "**demon**" slot, on the other hand, the system

will do nothing since this is the default. The inheritance specification will be explained below.

2.4. INHERITANCE

Relations defined among frames allow transfer of information from one frame to another. This mechanism makes it possible to store every piece of information at the most appropriate place.
In the FAIR system, inheritance may be controlled in two different ways:

- Local control can modify the inheritance of a slot by specifying the way how the slot may inherit information from other places and the way how the slot can be inherited by other places
- Global control can modify the inheritance along a particular relation

The information needed to control inheritance will be given in pre-fabricated frames.
The frame specifying local control can be filled-in by the user; if it is filled in, its name must be given in the meta-information associated to the slot (the inheritance of which is to be controlled). Actually, we may forbid the inheritance of the (value of the) slot or may determine the strategy of search for the next candidate (from which the lacking information will be tried to inherit) and finally we may give "hints" for the inheritance mechanism.
The global control of inheritance affects the information which may pass along a relation. The necessary information is given in the following frames:

```
frame: standard_relation_inheritance;
  excluded_slots:[];
  ex_condition:[];
  included_slots:[];
  in_condition:[];
end
```

The user may fill in this frame; then its name can be given as the value of the "**inheritance**" slot in the defining frame of the relation to be controlled.
The value of the slot "**excluded_slots**" (resp. "**included_slots**") can be a list of slot names. The specified slots will be excluded (included) when inheritance takes place if the appropriate conditions given in the "**condition**" slot hold, otherwise the

specification will be ignored by the system. Conflicts arising in this way will be resolved in FAIR by assuming that "exclusion" has greater priority.
For illustration consider the relation "similar_to", defined in our geometry example and the specification of inheritance along this relation:

```
frame: similarity_inheritance;
   included_slots:[alpha];
end
```

According to this specification the slot "alpha" and its value (and nothing else) can be inherited along the relation "similar_to".

2.5. WORLDS

Concepts of real world can be described from many different points of view. Frame-based systems usually supports this kind of gouping. In fact, FAIR also gives the possibility to partition frames into groups; these groups are called "worlds" in FAIR.
A world is simply a set of frames; all mechanisms (including inheritance) will always work inside a world.
Worlds can be connected to each other; the structure of worlds is a (rooted) tree. After initialisation the root will be automatically created. Then the user may define new worlds and may arrange them into the tree of worlds.
Worlds can be manipulated by fixed predicates. For example, the geometry example is described by means of one world called "geometry". This world can be defined as follows:

```
create_world(geometry,root,none).
```

As a result the world "geometry" will be defined as a son of the world "root", and will be leaved empty (i.e. none will be put into the new world automatically). In more advanced applications the user may define a new world and may specify what frames should be put from the ancestors to the newborn.
Navigation on the tree of worlds is possible by the "focus" predicate. For example, after executing

```
focus(geometry).
```

the actual world will be "geometry".

For technical reasons, FAIR will open two other worlds "rel_world" and "demon_world" immediately after initialisation. These worlds will be used to contain all relations and all demons, respectively. Worlds can be used for several purposes; for example it can be applied to simulate time dependency or to handle alternate hypotheses.

3. CONCLUSIONS

In this paper we described in some details the frame-language of the FAIR system. It should be stressed, however, that this frame-language is only one axis in the reresentation space of FAIR. The other component, the logic-based (rule-based) language, can be considered as "orthogonal" to the first one. Since the two components can be merged in an arbitrary way, the expressive power of the integrated language is very high.
The basic features of the integrated language of FAIR are as follows:

- Simple, and natural syntax
- Meta-information associated to parts of frames
  - demons, constraints, default e.t.c.
- User-definable relations
- Control of inheritance
- Horn-clause programming
- Structurability

REFERENCES

[1] Ait-Kaci,H., Nasr,R., LOGIN: A Logic Programming Language with Built-in Inheritance, J.Log.Prog., 3,(1986), 185-215.

[2] Brachman,R.J., Fikes,R.E., Levesque,H.J., KRYPTON: A Functional Approach to Knowledge Representation, Comput. 16 (1Ø) (1983),67-74.

[3] Clocksin,W., Mellish,C., **Programming in PROLOG**, Springer Verlag, (1981)

[4] Harmon,P., King,D., **Expert Systems**, Wiley and Son (1985)

[5] Results of Survey on Trends in Expert Systems in Japan, **Future Generations Computer Systems** 3 (1987), 17-36.

APPENDIX (a familiar example for computing area and perimeter of parallelograms)

Frames

```
world: geometry;

frame: polygon;
end

frame: tetragon;
   is_a:polygon;
   number_of_edges:[4];
end

frame: equiangle;
   alpha:[];
      meta_slot:alfa_meta;
         if_accessed_demon:[alfa_demon];
      meta_end
end

frame: equiline;
   property:[equiline];
end

frame: parallelogram;
   is_a:[tetragon];
   area:[];
      meta_slot:area_meta;
         if_accessed_demon:[area_demonfc];
      meta_end
   perimeter:[];
      meta_slot:perimeter_meta;
         if_accessed_demon:[perimeter_demon];
      meta_end
   height:[];
      meta_slot:height_meta;
         if_accessed_demon:[height_demon];
      meta_end
   side:[];
      meta_slot:side_meta;
         if_accessed_demon:[side_demon];
      meta_end
   base:[];
end

frame: rectangle;
   is_a:[parallelogram,equiangle];
end

frame: rectangle_1;
   base:[40];
   side:[10];
   instance_of:[rectangle];
end

frame: rhomb;
   is_a:[parallelogram,equiline];
end

frame: rhomb_1;
   instance_of:[rhomb];
   base:[10];
   alfa:[30];
end

frame: rhomb_2;
   instance_of:[rhomb];
   similar_to:[rhomb_1];
   base:[20];
end

frame: square;
   is_a:[rectangle,rhomb];
end

frame: square_1;
   instance_of:[square];
   base:[10];
end

frame: secans;
   90:[1];
   30:[2];
end

frame: goal_geo;
   condition:[];
   activity:[job_geo];
end

frame: dialog_geo;
   frame_name:[];
      meta_slot:question_for_fr_name;
         if_accessed_demon:[fr_demon];
      meta_end
   slot_name:[];
      meta_slot:question_for_sl_name;
         if_accessed_demon:[sl_demon];
      meta_end
end

frame: work;
   slot_name:[];
   frame_name:[];
end


world_end.

world: rel_world;

frame: similar_to;
   is_a:[relation];
   inheritance:[similarity_inh];
   inverse:[similar_to];
end
```

```
frame: similarity_inh;
  included_slots:[alpha];
end


world_end.


world: demon_world;

frame: fr_demon;
  is_a:[demon];
  activity:[fr_act];
end

frame: sl_demon;
  is_a:[demon];
  activity:[sl_act];
end

frame: alfa_demon;
  is_a:[demon];
  activity:[alfa_act];
end

frame: area_demon;
  is_a:[demon];
  activity:[area_act];
end

frame: perimeter_demon;
  is_a:[demon];
  activity:[perimeter_act];
end

frame: side_demon;
  is_a:[demon];
  activity:[side_act];
end

frame: height_demon;
  is_a:[demon];
  activity:[height_act];
end


world_end.
```

MPROLOG-rules

```
job_geo :-
    access_value(dialog_geo,frame_name,[FN]),
    access_value(dialog_geo,slot_name,[SN]),
    strieve(work(frame_name(FN),slot_name(SN))),
    access_value(FN,SN,V),answer(V) .

answer(V) :-
    V=[], write("I$'m unable to answer Your
    question."),
    nl;
    write(V), write(" the answer."), nl .

side_act :-
    access_value(work,frame_name,[FN]),
    access_value(FN,property,[I]),
    I="equiline", access_value(FN,base,[B]),
    strieve(parallelogram(base(B),side(B))) .

height_act :-
    access_value(work,frame_name,[FN]),
    structure(FN1,list,[FN,alfa(A),side(S)]),
    strieve(FN1),
    access_value(secans,A,[SEC]), H is S div SEC,
    strieve(parallelogram(height(H),side(S))) .

area_act :-
    access_value(work,frame_name,[FN]),
    access_value(FN,base,[B]),
    access_value(FN,height,[H]), A is H*B,
    create_frame(parallelogram,area,A,"o"), + .

alfa_act :-
    access_value(work,frame_name,[FN]),
    access_value(FN,number_of_edges,[N]),
    X is 180-360 div N,
    create_frame(equiangle,alfa,X,"o"), + .

perimeter_act :-
    access_value(work,frame_name,[FN]),
    access_value(FN,base,[B]),
    access_value(FN,side,[S]), P is 2*(S+B),
    create_frame(parallelogram,perimeter,P,"o"), + .

fr_act :-
   nl, write("Enter the frame_name!"), nl, read(FN),
   read_token(expression end), cut_input,
   create_frame(dialog_geo,frame_name,FN,"overwrite")
.

sl_act :-
   nl, write("Enter the slot_name!"), nl, read(SN),
   read_token(expression end), cut_input,
   create_frame(dialog_geo,slot_name,SN,"overwrite").
```

Trace	Effect
> focus(geometry)	
+ focus	
> run(goal_geo)	
> job_geo	
> access_value(dialog_geo,frame_name,FN)	
> fr_demon	FN : rhomb_2.
+ access_value	
> access_value(dialog_geo,slot_name,SN)	
> sl_demon	SN : area.
+ access_value	
> strieve(work(frame_name(FN), slot_name(SN)))	
+ strieve	
> access_value(rhomb_2,area,V)	
** inheritance of area	
> access_value(rhomb,area,V)	
> access_value(parallelogram,area,V)	
* if_accessed_demon	
> access_value(rhomb_2,base,B)	B : 20.
+ access_value	
> access_value(rhomb_2,height,H)	
** inheritance of height	
> access_value(rhomb,height,H)	
> access_value(parallelogram,height,H)	
** if_accessed_demon	
> strieve(rhomb_2(alfa(A),side(S)))	
** inheritance of alfa	
> access_value(rhomb,alfa,A)	
> access_value(parallelogram,alfa,A)	
> access_value(tetragon,alfa,A)	
- access_value	
- access_value	
- access_value	
> access_value(rhomb_1,alfa,A)	A : 30.
+ access_value	
** inheritance of side	
** if_accessed_demon	
** inheritance of property	
	S : 20.
+ strieve	
> access_value(secans,30,SEC)	
+ access_value	SEC : 2.
> strieve(parallelogram(height(10), side(20)))	H : 10.
+ strieve	
+ access_value(rhomb_2,height,[10])	
+ access_value(rhomb_2,area,[200])	area : 200.
> answer([200])	
+	
+ job_geo	
+ run(goal_geo)	

XRL: An experimental knowledge engineering tool for studying the different programming paradigms in AI

Balint Molnar
Central Research Institute for Physics,
Advanced Systems Department,
H-1525 Budapest 114, P.O.B. 49 HUNGARY

Mihai Barbuceanu
Institute for Computers and Informatics,
8-10 Miciurin, 71316 Bucharest 1 ROMANIA

I. Introduction

We would like to present a short overlooking of the XRL systems originally developed in Romania by M. Barbuceanu and his team. This system was implemented in a special dialect of LISP running on PDP-11 compatible machines. We combined our efforts and tried to port the existing system to a VAX compatible machine in order to exploit the standardized Common Lisp and the 32 bits architecture. As the time passed the XRL incorporated more and more interesting features and provided for uncountable important experiences validating the theoretical ideas. After having successfully ported to VAX compatible machines we began to work out a User's Manual and a Language Reference Manual and this work is in progress.

The XRL architecture shows some interesting properties e.g. the evolutionary self-enhancement and -developing philosophy of the system which is only an application of the four important aims promoted by the research. Namely, we would like to realize the next features:

- abstraction
- vividness
- declarativity
- enhancement.

Our main purpose is to develop an appropriate architecture supporting the above mentioned features. As it can be seen these requirements need a hybrid architecture in order to integrate many distinct paradigms. Moreover, the architecture should be able to evolve making sure that the system itself can be modified by using its own knowledge engineering tools i.e. the organization of the system should be reflexive.

XRL is a hybrid multi-layer architecture in which lower level tools such as structured objects, production rules or prolog are contained in the lowest layer. The middle level tools represents already an XRL specific layer including concurrent refinement of structured object and set oriented refinement. The upper layer incorporates the XRL specific approach to the construction and enhancement of knowledge processing tools or domain models, the self generation tool at

this level plays an important role extending the XRL network abilities relating to the structured objects and to the knowledge processors and applies the policy of XRL to itself, to its own structure.

II. The outline of the XRL architecture

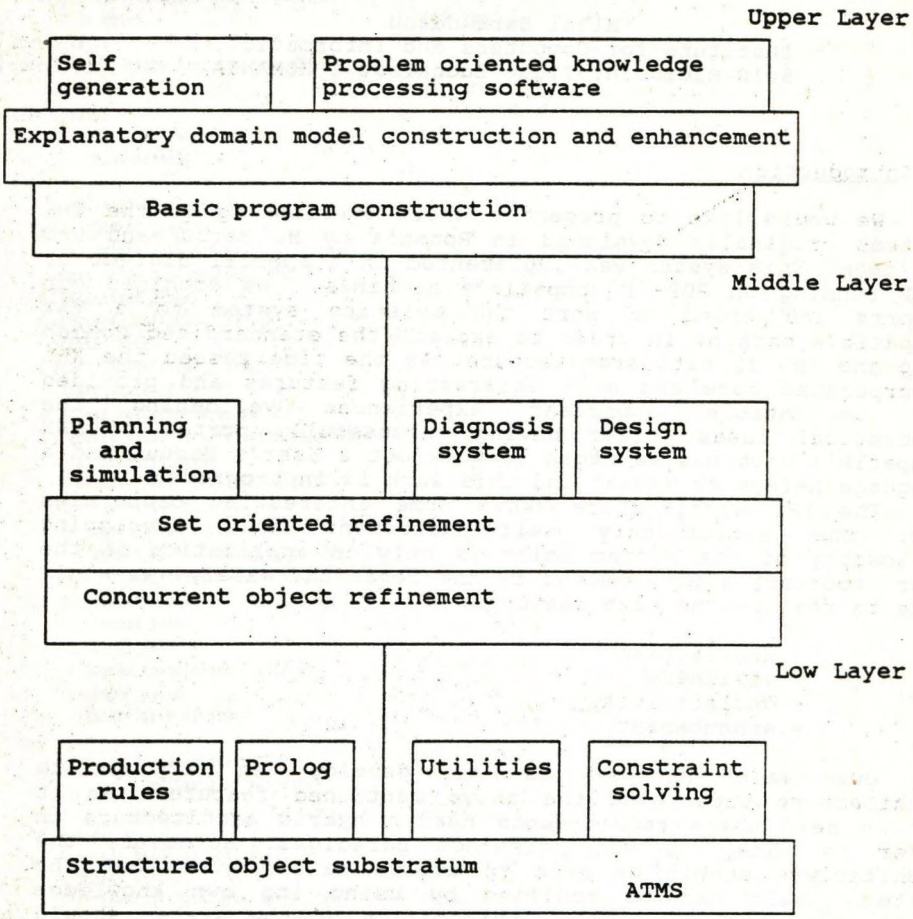


Fig. 1. The architecture of XRL

The figure 1. shows a structural diagram of XRL giving an overview about the important component of the system, the original illustration can be found in [Barbuceanu et al. 88]. We would like to explain here - only in rough details- these parts of the system.

1. The low layer

The low layer has the -so called- structured object substratum that contains the XRL structured objects being similar to the widely known frame concept in AI, or to the objects in the different object-oriented programming paradigms but they have a lot of specific and very advanced properties.

We can summarize the traditional attributes of the structured objects as the following:

(1) multiple inheritance
(2) meta-objects
(3) message passing with method combination
(4) active data (triggers)

As new aspects of the structured objects can be considered:

(1) the extended method combination abilities - XRL can combine the inherited methods as well as methods from the same lexical context
(2) the definition of XRL types for consistency checking and associative retrieval
(3) the special language defined for accessing the element of network consisting of structured objects
(4) ADT or abstract data type facility showing the relation between XRL and object-oriented philosophy and in addition providing for system interfacing and data handling capabilities

An example showing the most important character of the structured object in XRL - called unit here - will be presented:

```
[unit MyPreferredCar                    -the self slot holds the
 self [a *Unit supers (Car)             -unit meta description
         describe DescribeCar           -specifying inheritance
         draw DrawCar]                  -(slot supers) and message
colour silver                           -selectors with methods
engine [a TurboEngine]                  -(slots describeand draw)
passenger [a Nice Girl]]                -slots may also have meta
         [a *Slot*                      -descriptions holding
         chat-about ChooseSubject]      -selectors with methods
```

This example contains illustrations for descriptors such as "a Turbo Engine" or "a Nice Girl" that define ("describe") -in set theoretical sense- a set or subset of the certain general concept. In XRL, it is a notable notion that the set of the descriptors is not a predestined and unchangeable part of the system,however it can be flexible extended. The refinement semantics is strongly connected to the descriptors about which we will argue a little bit later.

A significant language peculiarity has been established for the XRL structured object network access, namely, a predicate testing the "is-a" relation between the units.

```
                                        -watch if a is-a b

(defpath ISA (a b)
 (prog(sups)
  (and(same-unit a b) (succeed t))      -succeed if a same as b
  (start a)                             -start from a
  (setq sups (step self supers))        -get the straightforward
                                        -supers
  (cond ((memq b sups)(succeed t))      -succeed if b is among
   (sups(apply 'split                   -them. Else split the is-a
     (mapcar '(lambda (x)               -path recursively on each
      (list 'isa x b))                  -of them
            sups)))                     -fail if no
      (t (fail)))))                     -straightforward supers
```

The primitives of the access language used are the subsequent:

start - sets the starting point of the path

step - steps one or more slots from the current instance

split - treats several paths in parallel returning the result of the first successful

succeed - returns from a successful path

fail - returns from a failed path.

As the above described illustrations shows, we can define -in the same manner- XRL types and ADTs (abstract data type). Who are interested in more details we can recommend them the next publications [Barbuceanu et al. 88], [Barbuceanu et al. 87].

In the existent AI (hybrid) systems, the production rules plays an important role providing for a simple tool acquiring a basic knowledge base pretty quickly. Because of this reason XRL contains a subsystem implementing the production rules and its interpreters. In despite of the simple rule based systems where the mechanism of the interpreters cannot be altered, here in XRL, the rule interpreters can be specified harmonising to the given domain (or model). The rules and as well as the rule interpreters represented in the same structured object format presented above.

The rule interpreters supply a message passing interface awarding the compilation and activation of the rule interpreters, this protocol allows the compilation, matching and application of rule. The rule interpreters permit several control strategies. Some are very simple, like "fire all rules once" or "finish after first successful firing". More complex ones use conflict resolution criteria (programmer or system supplied). Finally, an implementation of procedural production systems is also provided [Georgoff 86].

We present an example to illustrate the main ideas of the rule interpreters.

```
[unit RuleInterp-i                      -the meta unit RuleInterpMeta
 self [a RuleInterpMeta]                -provides the message
                                        -protocol
 args (delta)                           -lambda arguments of the RI
 vars ((temp -5)                        -initialized local variables
      (state frozen)
 rule-args (temp state)                 -arguments sent to the rules
 normal-rules (R1 R2 R3 ...)            -simulation rules
 init-rules (R-report)                  -status reporting rule
 term-rules (R-report)
start-rules (R-incr-temp-delta)         -increment temp with delta
control agenda                          -use conflict resolution
conflict-res best]                      -take "best" matching rule
```

The rule interpreter contains several rules that it controls, e.g.,there exists initialization rules which are carried out when the rule interpreter is activated, there exists termination rules on exiting from the rule interpreter, there exists normal rules which the formulated control strategy is applied for, start cycle and end cycle rules are triggered before and after each normal rule activation cycle. Procedural interpreters, which do not work in the usual match-solve-conflict-execute cycle have special slots describing the transition network which supervises the rule activation.

In addition to the above described rule-based subsystem, there does exist a Prolog architecture possessing the well-known backward chaining philosophy using Horn clauses implemented in the XRL object oriented manner. Prolog programs can access the object data base through the mechanism provided by the structured object language (invoked as function calls). The XRL involves such features as the above mentioned paths and associative retrieval, these are especially powerful and have replaced the need for a previous interface which treated structured objects as relational tuples (a method used in SRL [Wright and Fox 84]).

Constraints - in the sense of [Sussman and Steele 80] - appear as an independent XRL tool that can use the same general structured object and message passing mechanisms illuminated for the other tools. This tool allows the simultaneous use of several constraint networks. For each such network, there exists an object which describes the component constraints and their inter-connection and provides the message passing interface allowing the use of the network. This interface provides messages for network creation, setting initial values, value propagation, incremental value modification and instrumentation. Each constraint in part is represented as an XRL unit with slots for the value cells and for the value computation specification and with message types for activating value computation and handling conflicting situations.

Regarding the ATMS [deKleer 86a 86b 86c], it will be used in conjunction with the other XRL tools as a sophisticated cache which is able to avoid duplication of the problem solving effort. The ATMS is not implemented in the object oriented style due to both efficiency reasons and to the fact that we consider it as a tool of this substratum (like the

structured object language) which must be more stable than the higher level ones.

2. The middle layer

A powerful and widely used mechanism for bringing to bear the knowledge encoded in structured objects is instantiation, a procedure by which the knowledge contained in a generic object is employed to the construction of a similarly structured terminal object. In spite of the ubiquitous occurrence of instantiation in frame systems, few languages support explicitly this process. One of these rare case is LOOPS [Bobrow and Stefik 81] with its composite object, but even here only a rigid recursive mechanism is provided.

The middle layer of XRL delivers two tools that implement the frame instantiation. The formalized instantiaton process is called refinement. The first tool is based on interpreting structured objects as specifications of loosely coupled concurrent refinement processes, each process producing an instance of its generic unit. The second introduces a set theoretic language of descriptions which extends the structured object language, axiomatic the instantiation of the emerging notion of structured object and provides the machinery to implement the new instantiation concept on top of the former concurrent refinement tool.

The concurrent refinement system deals with the computational processes by which a structured generic object can be transformed into an instance of itself. For example, an instance of MyPreferredCar will have in the engine slot, instead of the initial generic unit TurboEngine, a more refined specialized instance of it such as:

```
[a TurboEngine
   power 120
   #cylinders 6
   #speeds 5].
```

This more refined (but not necessarily terminal) instance can be obtained by several computational processes. For example, one can take the TurboEngine unit and attempt to refine each of its slots. Alternatively, one can search in the knowledge base for a more refined instance of TurboEngine and place it in the engine slot. The first refinement process may be called expanding (the TurboEngine unit) and the second anchoring (the TurboEngine unit to an existing instance of it).

The concurrent refinement tool provides a framework for declaratively specifying and carrying out refinement processes of the above kinds. An essential aspect of the framework is that it allows several refinement processes to run in parallel. The issues of communication and synchronization are resolved by special primitives provided by the framework.

1) Refinement tasks. Any XRL structure to be refined is associated to a task by the framework. As an extension of the concept, beside units, arbitrary evaluable expressions can be also placed in slots and refined by evaluation.

2) System organization. The framework promotes the organization of the refinement processes into refinement systems.

3) Communication and synchronization. Conceptually, tasks not explicitly scheduled, however, are executed in parallel. Communication and synchronization are achieved by a form of communicating sequential processes which extends the path construct from the structured object level.

4) Control regimes. Beside the refinement described previously, other control regimes are supported. A form of dependency driven **undoing** allows the programmer to selectively modify a refinement network and to trace all affected slots.

The above outlined refinement process can be also seen as a process of incremental elimination of uncertainty. Initially, one knows that the engine slot will contain a TurboEngine, but there is a whole class of such engines which are implied by this description. Further refinement reduces this class to those with power=120, then to those also having 5 speeds, a.s.o. The process stops when the remaining level of uncertainty becomes acceptable.

The set oriented refinement tool formalizes the rules of this method. It extends the structured object language by maintaining descriptions formed with set operations (union, intersection and difference), - the someof **constructor** and the oneof **extractor** -, it provides a formal semantics of the resulting language based on the conception of description extension and it extends the concurrent refinement machinery to perform refinement specified in this manner. The formal semantics of the set language (called SODL) is described in [Barbuceanu et al. 87 and 88], the idea being somewhat similar to [Brachman and Levesque 84].

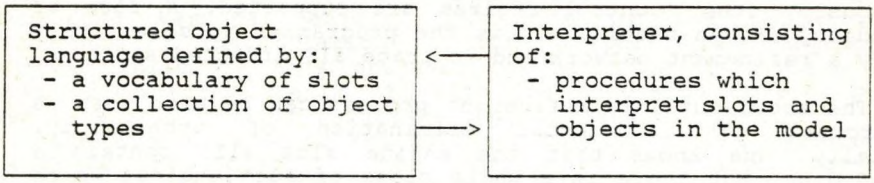
We have used these notions of refinement in several applications in design, planning and simulation and diagnosis. The experience we accumulated shows that refinement is an appropriate middle level allowing efficient implementation of several types of problem solving activities. Two such types of our experience show that can be fully and easily implemented are those described by [Chandrasekaran 86] as hierarchical classification and hierarchical design [Barbuceanu 85].

Considering the distinction made by [Clancey 85] between classification -in which the solution is one from a pre-existent enumeration and construction (where no such enumeration exists)- refinement appears as a good mechanism for the latter. This stems from the fact that refinement works by assembling partially specified pieces into higher order aggregates which do not have to be known in advance.

3. The upper level

The explanatory knowledge processor construction and enhancement endorses the construction and enhancement of a wide range of such processors, from low level domain independent ones such as production rule systems, to middle level ones such as the concurrent refinement one and further to task level ones such as design systems. We call all these sorts of processors domain models.

The theory and tools to be discussed in the next short section assume a certain characterization of domain models. From a representational point of view, they are assumed to be structured object languages characterized by certain vocabularies of slots and object types, together with an interpreter that is able to process the types of structured objects and slots defining the language. The next figure depicts this acceptance of the term.



The object oriented nature of domain models hides in fact a commitment to data-driven programming. The approach appears to be suitable to data-driven programming where relevant parameters of the programs are explicitly represented and accessible as data structures with possibly procedural annotations. The exact reasons for this will become clear later on.

The approach consists of two main steps. The first step uses: (1) a prototype version of the model, (2) a number of parameters of the model, (3) explicit specifications of the assumptions made about these parameters.

The parameters and assumptions, we can call them together explanatory structures, can be extracted from a data base which holds programmer defined or system derived explanatory structures. These elements form the base for re-formulating the model in a form suitable for the second step. Essentially, this form is functionally equivalent but it explicitly represents the dependencies between assumptions, parameters and the content of the model. These dependencies show how parts of the model depend on given assumptions and parameters.

The second step uses the re-formulated model as a base for a number of "semantic editing" activities which modify the assumption and/or parameters and propagate the effects of these modification on the model. The result is the production of the new models which work under the modified assumptions and/or parameters. A second result is the production of new explanatory structures from modifying the previous ones. These are archived in the explanatory structures data base. Semantic editing activities are possible because the re-formulated - or explained - form of the model explicitly links the relevant assumptions and parameters to the parts of the model which depend on them.

4.Remarks

Because of the lack of the space, we cannot go into further details, but both of the authors would send - with pleasure - a copy of the above mentioned reports about this topic to whom are interested in this theme and would ask us for a specimen.

References

1. Barbuceanu, M. (1985) An object centred framework for expert systems in CAD, in J.S.Gero (ed.), Knowledge engineering in CAD, North Holland 1985, 232-253.
2. Barbuceanu, M. , Trausan-Matu, S., Molnar, B. (1987) Integrating declarative knowledge programming styles and tools for building expert systems and their applications, Proc.7th International Workshop Expert Systems and Their Applications, Avignon, France 1987, EC2,1171-1196.
3. Barbuceanu, M. , Trausan-Matu, S., Molnar, B. (1987b) Integrating declarative knowledge programming styles and tools in a structured object AI environment, Proc. IJCAI-87, Morgan Kaufmann Pub. Inc., 563-568.
4. Barbuceanu, M. , Trausan-Matu, S., Molnar, B. (1988) The XRL2 Manual, in preparation.
5. Brachman, R.J., Levesque, H. (1984) The tractability of subsumption in frame based description languages, Proc. AAAI-84,Morgan Kaufmann Pub. Inc.
6. Chandrasekaran, B. (1986) Generic tasks in Knowledge based reasoning: high level building blocks for expert system design, IEEE Expert, Fall 1986, 23-30.
7. Clancey, W. (1985) Heuristic classification, Artificial Intelligence 27, 1985, 289-350.
8. Georgoff, M., Lansky, A., Bessiere, p. (1985) A procedural logic, Proc.IJCAI-85.
9. Bobrow, D.G., Stefik, M. (1981) The LOOPS manual, TR-KB-VLSI-81-13, Xerox Palo Alto Research Centre.
10. Sussman, G.J., Steele, G.L. (1980) Constraints - a language for expressing almost hierarchical descriptions, Artificial Intelligence 14, 1980, 1-39.
11. Wright, J.M., Fox, M.S. (1984) SRL 1.5 User Manual, T.R. C.M.U., Robotics Institute, 1984.

FLC: An Experimental Language for Designing and Implementing Frame-based Representation Features

László Kiss & Attila Farkas

Advanced Systems Department
Central Research Institute for Physics
P.O.Box 49 Budapest 1525 Hungary

ABSTRACT

This paper describes an experimental language called FLC, which has been designed to facilitate the design and implementation of frame-based representation languages. Using FLC, the user can define his own representation schemes in a declarative way so that this descriptive information can be used to generate efficient code. The pieces of code obtained in this way can be usefully incorporated when implementing a frame-based representation language. The code generation technique also enables the iterative modification of the representation language during knowledge base construction to better fit the needs of a particular application.

1. INTRODUCTION

The research area of knowledge representation has a long, complex history. During the past 20 years numerous representation languages have been designed. A characteristic feature of the beginning of this period was that many of the representation languages were built around one particular application area (e.g., Units [Stefik, 1977] for molecular biology or KRL [Bobrow & Winograd, 1977] for natural language understanding). Though this was beneficial to the specific application the language designer had in mind, it often resulted in the system's inadequacy for later use in somewhat different types of applications. The lack of widely useable representation languages often obliged AI projects to start by designing and implementing a knowledge representation language suitable for their particular application. This made the time spent with an application undesirably longer.

By the beginning of the eighties the drawbacks of this phenomenon had commonly been realised [Brachman & Smith, 1980] and attempts were made to overcome the difficulties. The two obvious solutions to the problem are to either construct far more general knowledge representation languages that are of use to a wider range of applications or to provide support for the design and implementation of representation languages so that the process can be shortened in time. There are a great number of representation languages demonstrating the useability and success of the first idea. In order to illustrate their approach, two such languages, RLL and SRL, will be described briefly in the next section.

The rest of the paper tries to contribute to the second approach, namely, the acceleration of the design and implementation phase of representation languages. More precisely, the paper deals only with *frame-based* representation languages [Minsky, 1975]. An experimental language called FLC is described, which allows the user to define his own (frame-based) representation schemes in a declarative way so that this descriptive specification can be used to generate efficient code. The pieces of code obtained in this way can be usefully incorporated when implementing a frame-based representation language. Finally, at the end of the paper some examples of how to use FLC are also given.

2. FLEXIBILITY AND REFLEXIVITY IN REPRESENTATION LANGUAGES

Much work has been done since the beginning of the decade to provide more general frameworks for representing knowledge in AI applications. One result of these investigations is that the architecture of truely flexible systems is necessarily hybrid, that is, these systems must integrate many distinct representational paradigms. Another key feature to flexibility is the reflexivity of the architecture, which means that a flexible system must contain in itself the capability to modify its components and its organization.

One of the earliest representation systems meeting the above requirements is the RLL system [Greiner & Lenat, 1980]. The lowest layer of RLL is a frame-based representation facility. Using this facility, RLL explicitly represents the components of representation languages in general and of itself in particular. Starting from the initial RLL environment the user can step by step modify any aspect of the system's operation (e.g., automatic inference mechanisms, fundamental access functions and control regimes) by modifying or creating units representing system components. In such a way RLL can be tailored to suit many specific applications.

Purely frame-based languages, which are the main theme of the paper, have evolved in a similar way in the last few years. They also have adopted some kind of reflexivity. They define a way of representation for the concepts relevant to the representation language itself (e.g., slot, relation and inheritance) and, additionaly, the fundamental access functions of the language are implemented in such a way that their operation is highly dependent on the part of the knowledge base representing these concepts. Manipulating this part of the knowledge base, the user is able to adjust the language to his particular needs.

An example for a frame-based language with reflexive features is SRL [Wright at al., 1984], which calls its representational units *schemata*. In SRL, the concept of a relation is represented as a schema. The initial set of relations contains only two system defined relations, *instance* and *is-a*, which are also represented as schemata. One of the most powerful features of SRL is that the user can define new relations that are specifically tailored to the needs of a particular application. To define a new relation, a schema with the name of the relation has to be created, and this schema has to be linked to the predefined *relation* schema by the *is-a* relation. The inheritance semantics of the relation, that is, the specification of what information is to be passed along the relation (e.g., which slots' values should or should not be inherited), can be given by filling in the slots of the schema (inherited from the *relation* schema through the *is-a* relation) in the appropriate way.

3. A CHARACTERIZATION OF FRAME-BASED REPRESENTATION LANGUAGES

Despite the long history of frame-based representation languages, there is still no agreement on many related topics. Certain areas (such as multiple inheritance, or inheritance with exceptions [Touretzky et al., 1987]) are still under intensive research. Another major source of variations is that in frame-based representation languages reflexivity is implemented in different ways and to different degrees. If we consider only those properties that are shared by all of the frame-based languages, we are left with a few common features. Based on these features the following characterization of frame-based languages can be given:

- In frame-based systems the information is indexed by the objects. While questions related to knowledge stored about a given object can be answered efficiently, it usually needs a lot of effort to answer queries like finding out which objects hold a certain property. (In hybrid systems, such questions can be answered easily by using, for example, backward chaining rules.)

- In addition to simply storing and retrieving information, frame-based representation languages automatically perform, as part of their assertion and retrieval operations, a set of inferences over the encoded information. Thus, the amount of information accessible in frame-based systems is larger than that explicitly encoded in frames. Well-known examples of automatically

performed inferences are inheritance, and inference methods that use descriptive information to maintain semantic integrity constraints [Fikes & Kehler, 1985].

4. OVERVIEW OF FLC

In [Greiner & Lenat, 1980] the authors liken RLL to a stop organ rather than a piano. By this analogy the stops of the organ correspond to the predefined representational parts (e.g., slots, inheritance, etc.) of RLL. Just as a stop organ can be made to sound like a piano by pulling and pushing the appropriate stops, RLL can be made to resemble a very wide range of representation languages, including, for example, KRL, OWL [Szolovits at al., 1977], and KL-ONE [Brachman & Schmolze, 1985].

Following the analogy, this paper proposes to assemble the piano from basic building blocks, rather than obtain it as the result of specialising a very general instrument. What makes this approach acceptable is that a mechanism is provided allowing the main modules of the piano to be built automatically from specifications describing the modules.

The FLC language has been designed to bring about the program illustrated by the analogy. FLC's architecture is based on the above characterisation of frame-based languages. The building blocks used by FLC to construct new representation schemes are the elements of a primitive object-centered data structure, which we call *stripped frames*. This data structure serves as the actual repository of data for the frame language to be designed. FLC accesses this data structure through its predefined access functions.

In addition to storing information, frame-based languages are also expected to perform various inferences over the encoded information. So, when designing a frame-based language, we also need some means of specifying what inferences it should perform. For that purpose, FLC contains a declarative sublanguage which allows the user to define "frame specific" inference schemes in the form of logic assertions. For example, the user can make assertions about the cases in which a slot should be associated with a certain frame. The frame specificity of FLC logic assertions is the result of restricting the way they can be defined. They can be built only from predefined clauses corresponding to calls of the access functions of stripped frames.

The purpose for designing FLC was to facilitate the design and implementation of frame-based representation schemes. In the following, we illustrate how FLC achieves this goal. Suppose we want to implement the access function named *get-slots* of a frame-based language. The first thing we have to do is making logic assertions describing when a frame should contain a certain slot. Once these assertions have been defined a query can be issued for finding the slots of a frame. In FLC, however, the effect of this query is not an "assertion guided" search for the slots. Instead, assertions are used to generate efficient code for finding the slots in a "direct" way. Using this piece of code, defining the *get-slots* function can not be a problem.

FLC is an extension to Lisp. It is comprised of a collection of Lisp functions that fall into three categories according to their role in FLC. The first set of funtions are the access functions of the data structure called *stripped frames*. The second set includes only two functions: one for asserting clauses in the database of logic assertions and another one for their retraction. The rest of the functions are those producing Lisp code based on descriptions in the database.

4.1. Stripped Frames

If we strip frame-based languages from inheritance, inverse link maintainance, integrity constraints, etc., we are left with a simple record-like data structure where one can retrieve only those facts explicitly asserted. We use the term *stripped frame* for a data structure with such properties. The construction of a more sophisticated data structure, e.g. a particular frame-based representational scheme, can be based on such a relatively primitive data

structure. The structural properties of *stripped frames* defined in FLC are very similar to that of ordinary frames: a stripped frame is made up of a number of slots, each slot having a number of values associated with it. Additionaly, slots can have slots, too, called facets. The difference between the two data structures lies in their access primitives. While in the case of stripped frames retrieval and assertion is done on a "what you get is what you stored" basis, in frame-based languages a good deal of inference (e.g., inheritance) is performed at assertion and retrieval time.

Every representation language has to provide for some kind of repository of data. Stripped frames serve as a simple information storage for frame-based languages implemented with FLC. The access functions of a frame-based language can be defined in terms of the access functions of stripped frames. It is this definition process that FLC tries to facilitate.

4.2. Logic Assertions

FLC contains a simple declarative language which allows the user to define "frame specific" inference schemes in the form of logic assertions. The specifications written in this sub-language have much resemblance to Prolog's Horn clause specifications [Clocksin & Mellish, 1981]. There is a major difference, however, in the way these specifications are used by the two languages. In Prolog, Horn clauses are fed into a general inference mechanism based on resolution. In FLC, the form of logic assertions is considerably restricted compared to that in Prolog: logic assertions can be built only from predefined clauses that can be associated with executable code (Lisp function calls). Exploiting this restriction FLC provides functions for generating Lisp code from the assertions.

The underlying ideas of the use of logic assertions in FLC can better be understood in comparison with how it is done in Prolog. The restricted form of FLC logic assertions can easily be imitated with Prolog Horn clause definitions. This imitation, which is also useful for giving semantics to FLC logic assertions, can also be viewed as a kind of representation for frames in Prolog, though very inefficient. By this representation the fact that the slot *s* of the frame *f* has the value *v* would be represented as

```
value(f, s, v).
```

For example, the assertions that mammals have four legs and dogs are mammals could be encoded as the facts

```
value(mammal, number-of-legs, four).
value(dog, is-a, mammal).
```

To be able to deduce that dogs have four legs, we need rules reasoning on the facts. Once the general rules

```
has-value(FRAME, SLOT, VALUE) :- value(FRAME, SLOT, VALUE).
has-value(FRAME, SLOT, VALUE) :- value(FRAME, is-a, V1),
                                 has-value(V1, SLOT, VALUE).
```

have been asserted we can ask the question

```
?- has-value(dog, number-of-legs, L).
```

and get the answer L = four. With this style of representation the encoded information can be used in a very flexible way. For example, it is easy to answer the question "What is that dogs have four of?" whereas frame-based languages do not support such queries.

In FLC's declarative sub-language assertions are represented in a way similar to the above example. For example, the above Prolog rules could be formulated as

```
((has-value f s <v>) :- (value f s <v>))
((has-value f s <v>) :- (value f 'IS-A <v1>)
                        (has-value <v1> s <v>)).
```

The first thing that strikes the eye is that FLC logic assertions are written according to Lisp syntax, that is, predicates are the first element of a list followed by their arguments separated with spaces. Another difference between Prolog and FLC logic assertions is that the latter can use Lisp variables in addition to constants and logical variables. This also contributed to the change of the simple Prolog syntax. Lisp variables and constants follow Lisp syntax, logical variables can be distinguished by the angle brackets enclosing them.

The use of Lisp variables in FLC logic assertions is more a limitation than a feature. The fact that the first two parameters of the *has-value* predicate are Lisp variables instead of logical variables indicates that these parameters have to be instantiated with either a Lisp variable or a constant (and not a logical variable) when a query using this predicate is issued. This implies that a query for finding all the existing frames with a certain property can not be issued, which is in accord with the object-centred nature of frame-based languages (i.e. with the fact that the encoded information is accessible via the objects). Lisp variables can be viewed as formal parameters to clause definitions. Only those Lisp variables which appear in the head of the clause can occur in the tail of a clause.

In FLC, logic assertions can be built only from predefined predicates. The basic predicates, in terms of which further predicates can be defined, are listed below with their respective arguments:

predicates	arguments
frame	frame-name
slot	frame-name, <slot-name>
value	frame-name, slot-name, <value>
facet	frame-name, slot-name, <facet-name>
facet-value	frame-name, slot-name, facet-name, <facet-value>

The arguments which are *not* enclosed in angle brackets can not be substituted with *uninstantiated* logical variables. That is, the first occurrence of a logical variable within the left hand side of a clause has to be in place of an argument enclosed in angle brackets. For example, in the above definition of *has-value* the logical variable <v1> appears in the term (value f 'IS-A <v1>) before it is used in (has-value <v1> s <v>).

In addition to the predicates listed above, FLC provides the standard Prolog predicates *not*, *equal* and *cut*.

2.3. How to Use the Assertions

FLC provides three functions named *find-all*, *find-one* and *prove* for utilizing the information stored in the form of logic assertions. In Prolog, the *findall* predicate is used to determine all of the terms that satisfy some goal. The Prolog goal *findall*(X, G, L) constructs a list L consisting of all of the objects X such that the goal G is satisfied. The *find-all* function in FLC takes arbitrary number of arguments. The first is a logical variable and the rest is a conjuction of goals to be satisfied. It returns a Lisp expression that, when executed, computes and returns a list of all of the solutions to the variable satisfying the goals. For example, the Lisp expression returned by the function call

```
(find-all <z> (has-value x y <z>))
```

collects all the values <z> which are associated, according to the above definition of *has-value*, with the *y* slot of the frame *x*. The symbols *x* and *y* are Lisp variables in the generated code and they are expected to be specified when the code is executed.

The pieces of code generated in this way can be enclosed in a function definition defining an access function of the frame language under development.

The function *find-one* is similar to *find-all* except that the code generated by it searches for only one solution. The *prove* function takes any number of goals as its arguments and the code produced by it returns a logical value according to the satisfiability of the goals.

2.4. Generating Code from Assertions

This section gives a rough outline of how the functions *find-all, find-one* and *prove* convert logic assertions to procedural Lisp specifications. The method of code generation can, in fact, be quite naturally derived, still writing the generated code in every particular case instead of making logic assertions would be a rather tedious task regarding the highly recursive nature of the problem. Furthermore, keeping the specification of our representation language in the form of logic assertions enables the iterative modification of the language during knowledge base construction to better fit the needs of a particular application.

The thorough compillation of FLC logic assertions into procedural Lisp specifications is made possible by three major restrictions on the language. These are the following:

- New predicates can only be defined in terms of some predefined predicates specific to FLC (frame, slot, etc.) and usual built-in predicates like *not* and *equal*.

- The use of logical variables in assertions is considerably restricted.

- The run-time definition of predicates is not allowed.

The code generating procedure, which works by a process of stepwise refinement, will be briefly illustrated in the case of the function call (find-all <y> g_1 ... g_n), where g_1, ..., g_n are goals and <y> is a logical variable that appears somewhere in at least one of the goals. The code returned by this function call is supposed to compute and return all the solutions to <y> that satisfy the conjunction of goals g_1, ..., g_n. During the computation, the set of solutions is gradually narrowing as the goals are processed one after the other. As soon as a goal proves to be unreachable, the procedure is over.

This computation scheme can be expressed in Lisp using the logical operation AND [Winston & Horn, 1984]. The first step of the refinement process produces the code

(AND p_1 ... p_n y),

where p_i denotes a yet unknown Lisp procedure for deciding the satisfiability of the goal g_i. The p_1, ..., p_n procedures are supposed to collect the solutions to <y> in a list assigned to the Lisp variable *y*.

The next level in generating the code is the further refinement of these procedures. Let t_{i1}, ..., t_{im} denote the tails of the clauses matching the g_i goal and assume for simplicity that these tails do not contain the *cut* predicate. The goal g_i is satisfiable if at least one of the conjunctions of goals t_{i1}, ..., t_{im} can be satisfied. The list of solutions to <y> is the concatenation of the lists of solutions found at each of the t_{i1}, ..., t_{im} goal expressions. Thus, the procedure p_i can be refined as

p_i = (PROGN p_{i1} ... p_{im} L),

where the Lisp construct PROGN is a simple control structure for sequential execution, p_{ij} denotes a procedure for deciding whether the conjunction of goals t_{ij} can be satisfied, and $\mathcal{L}$ is a logical value that is true if at least one of the goal expressions t_{i1}, ..., t_{im} is satisfiable. The p_{i1}, ..., p_{im} procedures are also supposed to append the list of solutions they found to the list contained in the above mentioned Lisp variable *y*.

The next step of the refinement process would be to further specify the procedures p_{i1}, ..., p_{im}. However, the task to be performed by these procedures is similar to what was done in the first step of the refinement process, namely, the decision of the satisfiability of a conjunction of goals. Thus, the code generating process can be defined as a recursive procedure. As the code for checking the satisfiability of the goals is generated, the alternatively changing AND and PROGN constructs become more and more deeply embedded into each other until, finaly, the goals with the predefined predicates are reached and the recursion stops.

5. EXAMPLES

5.1. The Inheritance of Slots in a Taxonomic Hierarchy

The first example shows how to define a function that collects and returns all the slots of a given frame x in a conventional IS-A hierarchy. Let us assume that in our frame-based language subclass and element links are established by the instantiation of the IS-A and INST slots, respectively. First the predicate *has-slot* will be defined to characterize the cases in which a slot is associated with a given frame. Note that the *has-slot* predicate below is made up of only those basic predicates listed in the previous section plus standard predicates like *not* and *equal*.

```
((inherit-slot-through-is-a x <y>) :- (slot x 'IS-A)
                                      (value x 'IS-A <z>)
                                      (slot <z> <y>)
                                      (not (equal <y> 'INST)))

((inherit-slot-through-is-a x <y>) :- (slot x 'IS-A)
                                      (value x 'IS-A <z>)
                                      (inherit-slot-through-is-a <z> <y>))

((inherit-slot-through-inst x <y>) :- (slot x 'INST)
                                      (value x 'INST <z>)
                                      (slot <z> <y>))

((has-slot x <y>) :- (slot x <y>))
((has-slot x <y>) :- (inherit-slot-through-is-a x <y>))
((has-slot x <y>) :- (inherit-slot-through-inst x <y>))
```

The definition of *has-slot* can now be used to produce Lisp code for finding the slots of a frame. This can be done by issuing the function call

```
(find-all <s> (has-slot f <s>)).
```

Suppose that the access function responsible for collecting all the slots of a given frame is called *get-slots*. Using the above function call, the Common Lisp definition of the *get-slots* function is as simple as

```
(DEFUN get-slots (f)
  #.(find-all <s> (has-slot f <s>))
).
```

where the #. construct, when encountered, causes the read-time substitution of the immediately following expression with the value of that expression.

In addition to finding the slots of a frame, the *has-slot* predicate can also be used to decide whether a given slot is associated with a given frame. The function call

```
(prove (has-slot f s))
```

returns the code to do the job.

5.2 Representing Default Information

Much of the real-world knowledge appears in the form of normative statements. These are statements that are usually true, or can be assumed to be true in the absence of contrary information. Many frame-based languages support the representation of such information and provide reasoning mechanisms for dealing with the cases violating the *default* assumptions. A commonly used *default* reasoning technique is that the value of a slot in a frame representing a set of objects is regarded only as a *default* value and is inherited only if a specific value for a particular individual is not known. In FLC this scheme can be implemented using the built-in *cut* (!) predicate. Supposing that the predicate *inherit-value* has already been defined, the default scheme can be expressed in the following way:

```
((has-value f s <v>) :- (value f s <v>) !)
((has-value f s <v>) :- (inherit-value f s <v>))
```

Sometimes it is convenient to associate a default value with a property (slot) in itself, rather than as the property of a set of individuals. That is the case, for example, when it is not easy to identify the set of objects having the property or, though the latter can be done, we do not want to represent the set in the knowledge base. If we want to have slots with their own default values in our representation language, first we have to decide how to represent that a slot has a default value. A very simple way of representation could be to create a frame with the name of the slot and put the default value in a slot called DEFAULT of this frame. Assuming this method of representation, the inferences comprising the default scheme could be encoded as:-

```
((has-value f s <v>) :- (value f s <v>) !)
((has-value f s <v>) :- (inherit-value f s <v>) !)
((has-value f s <v>) :- (has-slot f s)
                        (value s 'DEFAULT <v>)),
```

where the predicates *inherit-value* and *has-slot* are supposed to have been previously defined. The meaning of these assertions is that a frame inherits the default value of a slot if the slot is associated with the frame and a value for the slot in the frame can not be derived in any other way.

Schemes similar to the above default scheme are usually among the representational facilities provided by frame-based languages. Using FLC, however, one can define representation schemes far more specific to a particular application domain. Suppose, for example, that a biologist wants to create a knowledge base describing certain species of animals. He may find the number two a reasonable default value for the *number-of-eyes* slot. This domain specific knowledge can be directly "wired in" to the reasoning mechanism by appending the clause

```
((has-value f 'NUMBER-OF-EYES 'TWO) :- (has-slot f 'NUMBER-OF-EYES))
```

to the above definition of the default scheme. The advantages of encoding domain specific knowledge in the reasoning mechanism are that it increases the efficiency of information access and makes the representation language more suitable for the particular application. A

disadvantage of wired-in features is that they are not available for redefinition within the representation language.

ACKNOWLEDGEMENTS

We are grateful to Péter Krauth and Bálint Molnár for conversations which helped to clarify many of the ideas presented in this paper.

REFERENCES

Barbuceanu, M., Trausan-Matu, S., Molnar, B. Integrating Declarative Knowledge Programming Styles and Tools for Building Expert Systems. T. R. KFKI-1987-02/M, Budapest, Hungary

Bobrow, D.G., Winograd, T. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science* 1 (1), 1977, 3-46.

Brachman, R.J., Smith, B.C. (Eds.) Special Issue on Knowledge Representation. *SIGART Newsletter*, No. 70, February, 1980.

Brachman, R.J., Schmolze, J. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* 9 (2), 1985, 171-216.

Clocksin, W.F., Mellish, C.S. *Programming in Prolog*. Springer-Verlag, Berlin, 1981

Fikes, R.E., Kehler, T.P. The Role of Frame-Based Representation in Reasoning. *Communications of the ACM* 28 (9), 1985, 904-920.

Greiner, R. A Representation Language Language. HPP Working Paper HPP-80-9, Computer Science Dept. Stanford University, June 1980.

Greiner, R., Lenat, D.B. A Representation Language Language. *Proc. AAAI-80*. Stanford, CA, 1980, 165-169.

Minsky, M. A Framework for Representing Knowledge. In *The Psychology of Computer Vision*, P. Winston (ed.), McGraw-Hill, New York, 1975

Rychener, M. PSRL: an SRL-based Production Rule System. Carnegie Mellon Univ., Dec. 1984.

Stefik, M. An Examination of a Frame-Structured Representation System. *Proc. IJCAI-79*. Tokyo 845-852.

Szolovits, P., Hawkinson, L.B., Martin, W.A. An Overview of OWL, a Language for Knowledge Representation. MIT/LTS/TM-86, Massashusetts Institute of Technology, June 1977.

Touretzky, D.S., Horthy, J.F., Thomason, R.H. A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems. *Proc. IJCAI-87*. Milan 476-482.

Winston, P.H., Horn, B.K.P. *LISP*. Addison-Wesley, Menlo Park, 1984.

Wright, J.M., Fox, M.S. SRL/1.5 User Manual. Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA, 1983.

Sima, D., Kotsis, D., Kutor, L., and Tick, J.
Kandó Kálmán College of Electrical Engineering
Institute for Mathematics and Computer Science

REMOR AND REKNEL BASED KNOWLEDGE REPRESENTATION AND MANIPULATION

1. Introduction

In October 1987 our institute was commissioned by the National Committee for Technological Development (OMFB) to design and implement a knowledge based system for mathematics. After a review of several knowledge representation schemes we found that none of the systems were really meeting our requirements and therefore a different approach was adopted which we developed on our own and this is described below.

Knowledge representation techniques, especially those in extensive use since the late sixties, such as semantic networks, rule- or predicate-based logic, may reasonably be called as being of micro level, in the sense that they are composed of elementary objects and formal description of the relations between them, e.g. in the statement "father_of(Stephen,Steve)" written in PROLOG.

The representation of knowledge presented below is based on two objects of knowledge: on the so called remors

and reknels; it is a hierarchically structured and open-ended way of representation.

The remor and reknel based knowledge representations are of macro level, since the smallest unit of knowledge for consideration is a semantic unit related to a concept, just like the definition of the Riemann-integral: A function can be integrated, if the limits of the lower and upper Darboux sums are the same.

A macro level representation can be advantageously used in developing new electronic lexicons, textbooks, computer aided learning and teaching programs (CAL, CAT) or any novel electronic tools designed for knowledge transfer.

The systems implemented in line with the above principles seem to be closely related to the hypertext systems developed in recent years ([Co87],[Sh87]) and in some respects they can be considered as the outgrowth of our research activity on the field of CAL and CAT programs ([FS85],[GS88]).

2. Knowledge representation at a macro level

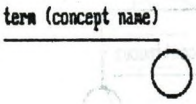
2.1. Knowledge morsels

In macro-level knowledge representation the so called *knowledge morsels*, which consist of the sum of all the knowledge associated with an individual concept, or more precisely the sum of all the knowledge that the originator of the knowledge morsel relates to the concept, are

considered to be of primary importance. The name of the concept is the term used as a identifier for the concept.

An example of a knowledge morsel is an entry with the associated paragraph in a lexicon, corresponding to a concept.

We shall use the following notation to denote a knowledge morsel

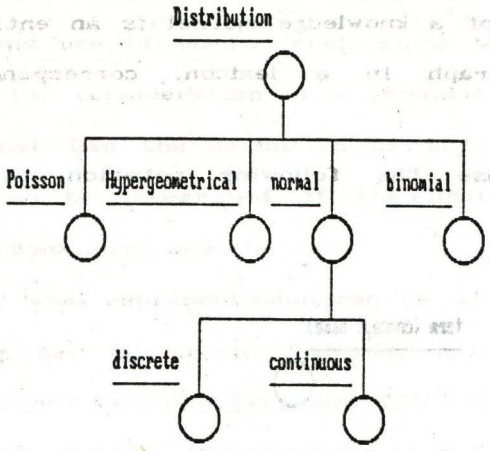


or in a simplified manner, if the concept name is not significant in terms of representation:

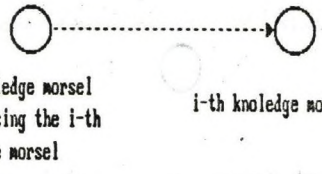
2.2. The remors

Crucial to knowledge is the fact that the individual concepts do not exist in isolation, rather, they are related to one another by significant relationships and relations.

Thus, for instance, concepts may be hierarchically related as is the case below



or also, in any knowledge morsel identifier may occur in other morsels as denoted below:

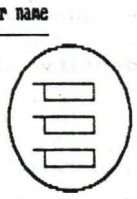


Now to reflect what has been shown above, it is high time that we extended the concept of knowledge morsel to all the relationships that relate a given concept to any other concepts, or rather, towards the relations that a person allocates to it to represent knowledge. Thus, instead of an isolated knowledge morsel, a *related knowledge morsel*, or

shortly *remor*, i.e. the union of this isolated knowledge morsel and the associated relations will be considered as the basis of knowledge representation.

2.3 Knowledge elements

Typically any remor will consist of the so called "*knels*", or *knowledge elements* denoted as:



Any particular knel is a semantically meaningful unit which is related to some aspect of a concept, such as the definition, or an attribute of a concept, or a list of examples in usage, etc.For example the definition, illustration and the scope of the Riemann integral, etc.

A knel maybe represented as textual, graphical, pictorial or even audio information, or a mixture of those.

It is important to emphasize that a knel may also be a procedure, such as finding the numeric integral using Simpson's method.

2.4 Related Knowledge Elements (reknels)

In this case it is possible that in any knowledge

element the identifier of another knowledge element or another remor may occur, or also, that the individual knowledge elements make up a hierarchical structure. Being aware of all that above, we should consider a *reknel* (*related knowledge element*) - i.e. a knowledge element with all its relations and references - as the basic unit of of knowledge representation.

To sum it up: remors considered to be the expansion of concepts, which in turn identify the relationship of a given term to any other terms. Remors will always consist of reknels, whereby each reknel will contain the references to individual knowledge elements related to the concept and the relationship of a knowledge element with other local or global knowledge element, besides the references to other remors, of the knowledge element in question.

It is important to note here that the representation described here has two crucial features, viz., the isolated knowledge elements and morsels, and the relations and references among them. From this point of view the representation of knowledge described here is to be regarded as a hierarchical hypertext system.

3. Manipulation on Knowledge

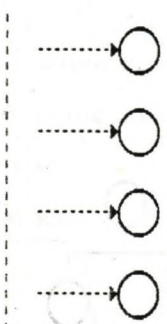
For the data structure outlined a number of knowledge manipulations can be defined in order to allow for the utilization of the same macro level knowledge base to meet

quite different needs, such as an electronic lexicon or a CAT or a CAL program. The type of application will be determir.ed by type of knowledge manipulation to be defined and implemented. Now it must be obvious that the above ways of representing knowledge offer new tools that surpass traditional methods for knowledge transfer by integrating the functions of classical tools and by providing new functions (e.g. functions that enable the use of procedures supporting rote learning and assessing knowledge level, etc.)

Below you will find a summary of the main functions contemplated.

3.1 Direct Access of Knowledge

This function enables the user to directly access the remors by their identifiers by adequate presentation of reknels.



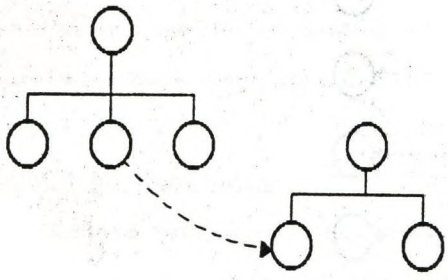
As a default option this function will enable the user to look up sequentially the reknels related to a remor just as the case is in reading through the entries related to a key term in a lexicon. A further option is to implement direct access of reknels by identifier.

It should be pointed out that electronic display of information has many new ways to utilize, one need not write out a reknel in full length on the screen, he may just as well decide to present parts of the text, or figures one after the other, some highlighted, some blinking, etc.

3.1.1. Supplementary Functions to Direct Access of Knowledge An Overview of Concept Relations

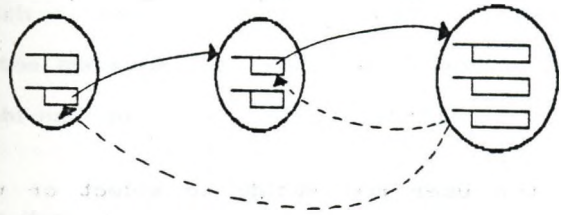
This function enables the user to learn the environment of the term searched in terms of concepts, to review the main and associated or basic concepts related to the term, and its further references.

Example:



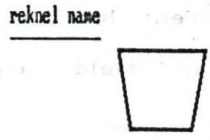
Following a reference chain

Should it occur that following through a chain of pointers or references, while accessing the reknels (or remors) in the knowledge base, a user finds a concept that he is less or not familiar with at all, he will be allowed to jump directly from that point to the remor or reknel of the concept in question, of course in a chained fashion so that he can also return to the original point going either straight back or by visiting each link in between.



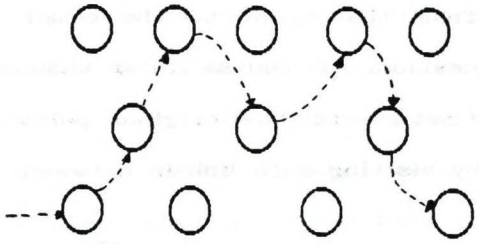
The Use of Procedures

Since reknels may be procedures the user is naturally free to use them to solve his problems, e.g. to identify the roots of an equation using some numeric method. The notation of a procedure reknel is

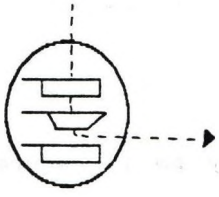


3.2 Supplementary CAL functions

Stored remors and reknels can be directly used for assembling computer aided teaching (CAT) programs. A course will be designed by defining the remors and reknels required into a chain.

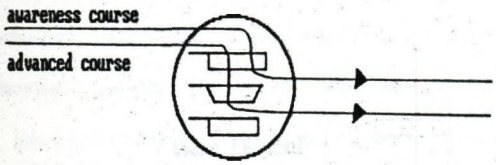


Of course the user may decide to select or unselect a reknel within a particular remor.



On the given knowledge base several electronic courses (CAT programs), each of different level to choose from, may be defined in a particular field, e.g. in Probability Calculus.

For example:



Electronic CAT courses will consist of a chain of remors and reknels selected and presented for the particular field, which can be supplemented by opportunities to memorize the material, to test the student's progress and to help him through problem-solving procedures.

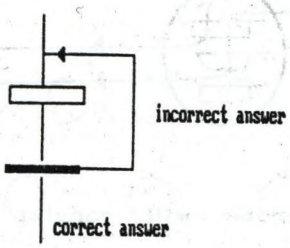
Supporting Memorizing or Rote Learning

By designing an efficient way of presentation, such as delaying and repeating parts of the materials on display, it is easy to implement supportive methods for memorizing knowledge.

Testing progress in obtaining knowledge

On any electronic course defined properly the user may include a test at any junction notation of a test.

Depending on the test results the presentation of the course material can be modified, see below



Skill acquisition

Tests may be designed and prepared by planned application of problem solving procedures (e.g. define the roots of a particular equation, etc.), or by calling for certain problem solving activities (e.g. supplying the definition of variables in a segment of a program written in PASCAL).

3.3. Relation between individual access modes to knowledge

Of course the access modes discussed should make it possible to swap between them, in other words, when someone is in a particular course, it should be possible for him to interrogate the knowledge base to learn about concepts he does not yet know and return to the original path of learning designed for him. Similarly, he may find it necessary to access the description of some terms while engaged in solving a particular problem.

At the same time, however, there must be an option to "backspace" on the route covered so far, right back to the starting point (say to the point of departure in the electronic lexicon), unless this starting point is too "far" (from a technical point of view it is desirable to keep a record of a given number of the most recent steps: this, however, is not in contradiction with any educational logic either, as there is hardly any user who could keep in mind where he set out for his "detour" and even less so, the points he visited in between, after he had meandered 10 to 20 steps off mainstream.)

It is one of the fundamental objectives of the system to be expandable.

CAT programs with fairly advanced "intelligence" can be implemented by inserting tests of this nature.

4. CATCHME knowledge base manipulation system

For the design and implementation of a subset of remor and reknel based knowledge representation and the knowledge manipulation options outlined above, a research program on a grant received from the National Committee for Technological Development (OMFB) was started at the Institute of Mathematics and Computer Science of Kandó Kálmán College (KKVMF). The name of the product is Computer Aided Tutorial Courses for Higher Mathematics Education.

In the project a good use was made of the experience gained in developing and applying TEACHSOFT, a package written to support teaching mathematics to undergraduates. TEACHSOFT was originally written by the staff of the Institute of Mathematics and Computer Science of the KKVMF in conjunction with several tutors from other colleges and universities in Budapest lead by D.Sima and I.Fenyö. The objective of the software package was defined as to provide new tools for teaching college level mathematics by making use of computer aided lecturing and computer aided practical work. TEACHSOFT, which is being sold to some more users in and outside Hungary, has been in use with several universities since.

CATCHME knowledge management system requires an IBM PC or compatible environment. The project itself shows some similarity to hypertext systems of which it is the closest to SAVAN ([De84]), an electronic lexicon and HyperCOSTOC, a CAT program developed at the university in Graz, Austria. In comparison with the systems referred to, CATCHME knowledge management system boasts with its remor and reknel based hierarchical knowledge representation, its approach to take procedures as knowledge elements and its large variety of knowledge manipulating functions as novel features.

References

[BF83] **A.Barr, E.A.Feigenbaum:**
The Handbook of Artificial Intelligence I.-III.
HeurisTECH Press, Stanford, 1982-83.

[Co87] **J.Conklin:**
"Hypertext: An Introduction and Survey"
Computer, Oct. 1985, pp. 15-30.

[De84] **J.L.Desalles:**
SAVANT: l'enseignement assisté par télématique dans la formation des ingénieurs de l'ENST
L'Écho des recherses N°117, 1984.

[FS85] **I.Fenyő, D.Sima:**
Ein didaktischer Versuch zum Unterricht der Mathematik an technischen Universitaeten und Hochschulen
Jahrbuch Überblicke Mathematik, 1985 pp. 139-142.

[Fo84] **R.Forsyth:**
Expert Systems. Principles and Case Studies
Chapman and Hall Ltd., London, New York, 1984

[Hu88] **F.Huber:**
Hyper-COSTOC: A Computer-Based Teaching Support System
BEYOND NUMBER CRUNCHING Third Austrian-Hungarian Informatics Conf. 1988. pp. 29-43.

[GS88] **A.György, Zs.Lukács:**

"Teachsoft" Educational Program Package

Sixth International Congress on Mathematical Educatior

Budapest, 1988.

[Ni82] **N.J.Nilsson:**

Principles of Artificial Intelligence

Springer Verlag, 1982.

[Sh87] **B.Shneiderman:**

Designing the User Interface: Strategies for Effective Human Computer Interaction

Addison-Wesley, Reading, Mass. 1987.

[Wa86] **D.A.Waterman:**

A Guide to Expert Systems

Addison-Wesley Publishing Company, 1986

DDL and DDS: A Dialogue Design Language and System for (Prolog) Expert Systems.

János Aszalós, Computing Applications and Service CO.
Postal Address: 1502 Budapest 112. P.O.B. 146.

Abstract

Expert systems are basicly interactive ones. Interactivity can be approached on many levels and from many points of view. The paper presents DDS as a tool and DDL as a language for organizing the dialogue flow of some dialogue-patterns used in several expert systems. Our viewpoint is syntactic.

DDS is implemented in Prolog and (now) is used for Prolog-based Expert Systems. It is an interactive one itself with some built-in knowledge about dialogue.

Introduction

This paper proposes an approach to dialogue-engineering. In the past 20 years many efforts have been made to investigate the syntactic, semantic, and psychological,

vi/ As our ES-s are /or would be/ implemented /now/ in Prolog, the DDS, which is also a Prolog application, has to compile /now/ the DDL sentences into Prolog-sentences. These restricitons would be lifted in the future.

2. DDL syntax

The highest syntactic unit in DDL is the DP which is composed of dialogue-elements /DE/; DE-s are in turn composed of dialogue-atoms /DA/.

A DA is either a statement /S/, or a question /Q/ or a command /C/, optionally preceeded by one or more statements /S^*/.

A somewhat simplified syntax-definition for DA:

<DA> ::= <S>|<Q>|<C>.

<S> ::= <sentence-body> <full stop>.

<Q> ::= <sentence-body> <question-mark>.

<C> ::= <sentence-body> <exclamation-mark>.

<sentence-body> ::= {sel-relname: relation-name, sel-qual:

qualificator, sel-mod: modificator,
{sel-fill: filler}}

All elements except the relation-name are optionals. Relationnames, qualificators and modificators are words, predefined by the expert and/or by the knowledge engineer, together with the possible abbreviations, radicals and synonyms. Fillers are free strings, not containing the other categories. None of the four categories /relationname, etc/ may contain <fullstop>,<question-mark> or <exclamation mark>.

The applicable /natural/ language is not determined by the syntax.
DA-s are indexed by the initiator, which is either the user /n/ or the system /s/. For example, $(S^*S)_n$ means a DA composed of one or more statements originated by the user /input for the ES/; $(S^*Q)_s$ is a question originated by the ES /output/ which can be preceeded by some statements. The relationname, the qualificator and/or the modificator of a DA can be represented by formal parameters, e.g.:
$(S)_s$: "There is a strong indication for X",

3. the use and the special features of DDS are presented. We summarize our experiences and future plans in the Conclusion.

1. Preconditions

i/ There are four individuals who come into picture in this context:

- the expert,
- the knowledge engineer,
- the computer system /the ES/, and
- the user of ES /probably distinct of the expert/.

ii/ The dialogue is carried out between the user and the ES, and it follows some "patterns". /The concept of DP is not to be detailed here; it is similar to the concept of conversation graph in [Hägglund, 80] or to the frames in [Bobrow, 77]/. The DP-s depend mainly on

- the problem classes to be handled by the ES,
- the problem-solving strategies,
- the knowledge of the ES,
- the model of the ES and that of the end-user as

they exist in the mind of the expert.

iii/ There are but a few DP-s for an ES, and they are known for the expert. /Some general information about DP-s are known for the DDS as well./

iv/ The expert wants to describe his knowledge about DP-s and wants to communicate it to the ES in a flexible manner just as his expertise. Therefore, he would welcome a language and a software tool for this purpose: these are offered him in DDL and DDS.

v/ The language must be rich enough to represent all the necessary elements and constructors for a restricted set of DP-s, but must be simple enough for an expert, say in medicine, who is unskilled in using formal languages. Therefore, he would likely accept the help of an interactive and self-explaining system. /The "restriction" mentioned above is the consequence of the restriction on the topics: we want to formalize ES-dialogues and not those of, say, Shakespeare-dramas./

methodological, organizational etc. aspects of this kind of engineering. These investigations are carried out either on sentence-level, or on the level of mutual understanding, or on the level of the whole conversation, etc.

Our approach focuses mainly on syntactic problems (together with some semantic considerations) on the level of the dialogue-patterns (DP). A DP is a linked set of Dialogue Elements (DE, see later), that form one or more paths possibly with cycles. The path to be followed during an actual dialogue is defined by the syntax of the user input at each DE, and by the knowledge incorporated in the DP, which interprets the user input, organizes the output and the transition to the next DE in the path.

The "output" of this work are a language definition for organizing the dialogue (DDL: Dialogue Design Language) and an interpreter and compiler for the language (DDS: Dialogue Design System).

(The work was carried out under the control of MESD - Methodology for Expert System Development - based on a concept of the "activity" elaborated in view of the cognitive psychology, system-theory and AI: see

[Aszalós, Gergely, 83]. Due to this support our work has its methodological, psychological and organizational fundaments of its own.)

Present expert systems usually offer various tools and languages for describing the knowledge in form of rules, nets, etc., but few of them /e.g. [Bateman, 83]/ present similar tools for defining the necessary dialogue-flow for cooperative problem-solving. The purpose for developing DDL and DDS is to add the corresponding new element to the building-kit of our ES-development project. As usual, the dialogue-frame of an ES is wired into the system or it is implemented as a subsystem by the programmer. Following our methodological considerations /which are not to be detailed here/ the responsibility of organizing the dialogue falls upon the expert and the knowledge engineer, just as that of defining the knowledge. Therefore, DDL and DDS is for the use of the expert, who is aware of the necessary dialogue-patterns but is unaware of the programming requirements.

The paper is organized as follows. In 1. we describe some preconditions and restricitions for using DDS. In 2., the syntax of DDL is illustrated by some examples. Then, in

where X would be turned into a name for an illness during the run-time.

A DE is a sequence of DA-s with alternating index. For example, $(Q)_n$; $(S^*S)_s$; $(S)_n$ is a typical DE: the user puts a question $/(Q)_n/$, then the system answers $/(S^*S)_s/$ and finally the user acknowledges or refuses the answer $/(S)_n/$.

DE-s are represented by frames. A frame in this context is a list of the following slots to be filled by the expert and/or by the knowledge engineer:

NAME:
INPUTi: /i=1,2.../
OUTPUTi: /i=1,2.../
TESTi: /i=1,2.../
GLOBVARS:
LOCVARS:
USE:
DEF:
CONTROL:

} declarations

Each DE has a unique NAME. LOCVARS can be accessed inside of the frame only; GLOBVARS inside and outside of the frame. The sequence of the INPUT and OUTPUT /and probably other/ procedures are determined in CONTROL. If there is a need for performing extra procedures or functions for elaborating the parameters, storing some intermediate results, printing, etc., the expert can insert the call for these procedures or functions into the CONTROL or TEST slots, but previously he must declare their names in USE or define them in DEF. There is a set of standard procedures and functions which can be used without declarations, e.g. logical functions, arithmetic procedures, string manipulation, etc.

An example for a menu-driven DE:

```
NAME: fever.
OUTPUT1: "Is the patient in fever?
          1./ no,
          2./ subfebrilis,
          3./ febrilis."
INPUT1 : PN, [PN is a standard variable for
          positive integers]
```

```
TEST1     : PN ≤ 3.  [All TEST-s are interpre-
                      ted as logical functions]
OUTPUT2   : "OK".
OUTPUT3   : "NOT NUMERIC VALUE OR OUTSIDE OF
            THE SCOPE. ENTER AGAIN!"
GLOBVARS  : X.
CONTROL   :    OUTPUT1;
            A: INPUT1;
               IF TEST1 THEN OUTPUT2, X:= PN,
               GOTO B; IF NOT TEST1 THEN
               OUTPUT3, GOTO A;
            B: END.
```

CONTROL represents a program in DDL which can be easily composed even by unskilled users /say, by an expert in medicine, geology, etc./.

A DP is a net, composed of nodes and directed arches, with one START and one END node. The nodes represent DE frames /one-one correspondence/; the arches represent "transition conditions" between two nodes.
One node may belong to several DP-s. The messages between DP and its nodes are transferred via global variab-

les.

The specification of a DP contains the following items:

. NETNAME /must be unique/,

. declarations /see DE description/,

. list of node-names, together with their "preconditions",

. TRANS: transition conditions between nodes.

To avoid redundancy, the performance of the nodes are specified outside of the DP-s. Inside of the DP-net, only their names and the necessary elaboration of the appropriate global parameters are described. /This is the task of the "preconditions"./

To illustrate this point, we would specify a partial DP corresponding to Fig. 1.

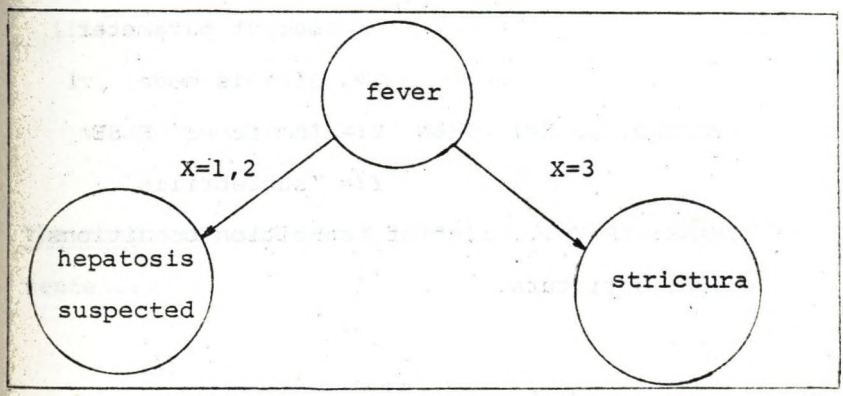


Fig. 1.

The nodes correspond to DE-s; the arches are indexed with the possible values of the "fever" node global variable X /see the previous section example/.

Specification of the DP "anamnesis":

```
NETNAME: anamnesis.
GLOBVARS: X,Y
NODE: START.
NODE: fever.
TRANS: IF X=1 OR X=2 THEN GOTO hepatosis
       suspected;
TRANS: IF X=3 THEN GOTO strictura
       [The last 2 conditionals represent the
        transition conditions corresponding
        to the arches.]
NODE: hepatosis suspected [Y is the input-
                           output parameter
                           of this node]
PRECOND: IF X=1  THEN  Y:= 'no fever' ELSE
                       Y:= 'subfebrilis'.
TRANS: IF Y=...[list of transition conditions]
NODE: strictura.
. . .
```

Multi-level DP-s

The CONTROL slot of a DE may contain a call for a DP /"call <DP-name>"/. Thus a dialogue represented by a DP may fire several sub-dialogues, which in turn may fire again sub-sub dialogues, etc. Thus a multi-level dialogue can be organized and controlled by DDS. This is necessary for ES-s because of the hierarchical structure of the ES dialogues. Recursivity is allowed.

3. DDS: a Dialogue Design System

DDS is composed of five parts:

i/ compiler,
ii/ run-time procedures,
iii/ knowledge-base,
iv/ house-holding procedures,
v/ self-explanation subsystem.

The compiler translates the DP-s and DE-s into Prolog sentences.

More precisely, it

a/ creates special data structures /=Prolog statements/ for each GLOBVAR, LOCVAR, NODE and NETNAME,

b/ translates the DDL sentences /=INPUT, OUTPUT, TEST declarations, preconditions and transconditions/ into Prolog procedure calls,

c/ creates the necessary Prolog procedure definitions, e.g. those corresponding to the CONTROL slots.

The task of the run-time procedures is

a/ to handle the created data structures,

b/ to elaborate the input-output parameters,

c/ to select the next DP or node to be processed, according to the transition-conditions.

The changeable knowledge-base of DDS /not yet ready/ would contain some predefined DP-s /a QA pattern, explanation-pattern etc./ together with some procedures for the knowledge base management.

The house-holding procedures serve for storing, restoring, deleting, replacing etc. functions.

The self-explanation subsystem offers the possibility of easy use of DDS for beginners, and contains a simple HELP subsystem as well.

As DDS is an interactive system, it is based on a two-level DP structure. In its last version, it uses all the features of DDL as well.

Conclusion

DDL and DDS offer a restricted but easily applicable set of tools for dialogue engineering. In their next state of development they would present the following new features:

i/ a graphic language for DP definition,
ii/ the knowledge base for basic DP-s,
iii/ an "intelligent" natural language interpreter,
iv/ Prolog-independence.

DDL and DDS have been used for the development of an ES for gastroenterological diagnosis, and for the last version of DDS itselt. Both are developed in SIEMENS-BS2000, but will run in microcomputers.

References

Aszalós, Gergely 83: Aszalós J. and Gergely T.: Activity-based Approach to Expert Systems and to their Knowledge, in Proceedings of ISAI Conference, Leningrad, 1983. Pergamon Press, /to be published/.

Bateman, 83: Bateman, R.F.: A translator to encourage user modifiable man-machine dialog, in Designing for Human-Computer Communication /Sime & Coombs eds/, Academic Press, London, 1983.

Bobrow, 77: Bobrow, D.G.: GUS, a Frame-Driven Dialog System. Artificial Intelligence, Vol. 8, no. 2, apr. 1977.

Hägglund, 80: Hägglund, S.: Contirbutions to the Development of

Methods and Tools for Interactive Design of Applications Software. PhD dissertation, Linköping University, 1980

NeurFrame: Simulating neural nets with MPROLOG

János Rácz
SzKI Computer Research and Innovation Center
H-1015 Budapest, Donáti u. 35-45.
18 Januar 1989

Abstract

Artifical neural net models have been studied for many years in the hope of achieving human-like performance in the fields of speech and image recognition. These models are composed of many nonlinear computational elements operating in parallel and arranged in patterns reminiscent of biological neural nets. Computational elements or nodes are connected via weights that are typically adapted during use to improve performance. There has been a recent resurgence in the field of artifical neural nets caused by new net topologies and algorithms, analog VLSI implementation techniques, and the belief that massive parallelism is essential for high performance speech and image recognition.

This paper provides an introduction to the field of stucturing the neural nets to solve some image processing. I use an initial structure to define the neural net topologie. I simulate the neural net models with MPROLOG ..

INTRODUCTION

Artificle neural net models or simply 'neural nets' go by many names such as connectionist models, parallel distributed processing models, and neuromorphic systems. Whatever the name, all these models attempt to archive good performance via dense interconnection of simple computational elements. In this respect, artifical neural net structure is based on our present understanding of biological nervous systems. Neural net models have greatest potential in areas such as speech and image recognition where many hypotheses are pursued in parallel, high computation rates arre required, and the current best systems are far from equaling human performance. Instead of performing a program of instructions sequentially as in a von Neumann computer, neural net models explore many competing hypotheses simultaneously using massively parallel nets composed of many computational elements connected by links with variable weights.

Computational elements or nodes used in neural net models are nonlinear, are typically analog, and may be slow compared to modern digital circuitry. The simplest node sums N weighted inputs and passes the result throught a nonlinearity as shown in Fig.1.

The node is characterised by an internal threshold or offset and by the type of nonlinearity. Figure 1 illustrates three common types of nonlinearities; hard limiters, threshold logic elements, and sigmoidal nonlinearities. More complex nodes may include temporal integration or other types of time dependencies and more complex mathematical operations than summation.

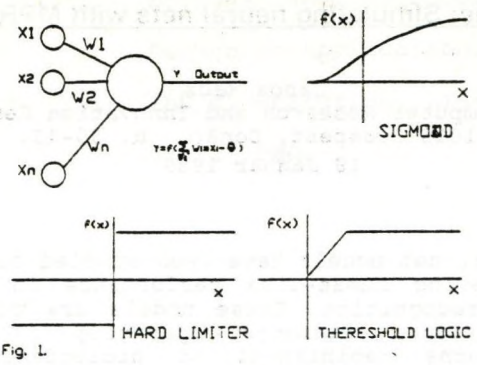


Fig. 1

Neural net models are specified by the net topology, node characteristics, and training or learning rules. These rules specify an initial set of weights and indicate how weights scnould be adapted during use to imptove performance. Both design procedures and training rules are the topic of much current research.

Work on artifical neural net models has a long history. Development of detailed mathematical models began more than 40 years ago with the work of McCulloch and Pitts [6], Hebb [2], Rosenblatt [10], Widrow [14] and others [9]. More recent work by Hopfield [3,4,5], Rummelhardt and McClelland [11], Sejnowski [13], Feldman [1], Grossberg [8] and others has led to a new resurgence of the field. This new interest is due to the development new net topologies and algorithms [3,4,5,12,1], new analog VLSI implementation techniques [7], and some intriguing demonstrations [13,5] as well as by a growing fascinacion with the functioning of the human brain. Recent interest is also driven by the realisation that human-like performance in the areas of speech and image recognition will require enormous amounts of processing. Neural nets provide one technique for obtaining the required processing capacity using large numbers of simple processing elements operating in parallel.

NeurFrame: Simulating neural nets with MPROLOG

NeurFrame (an empty frame for simulation of neural nets) has the following aim:

to give an empty frame for simulation of all kinds of neural nets with

- dynamical modification of stucture, model and algorithm
- possibility for free experimentation
- possibility for meta-frame declaration:
  - heuristics for neural net modification
  - learning algorithms
  - possibility for genetic algorithms

The System Architecture is shown on Fig. 2.

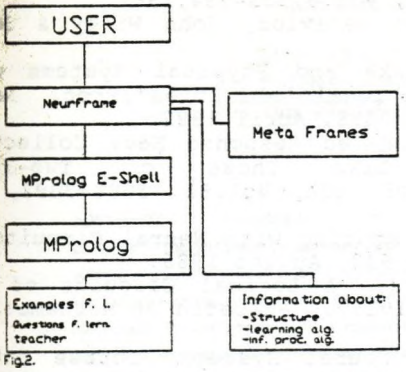


Fig.2.

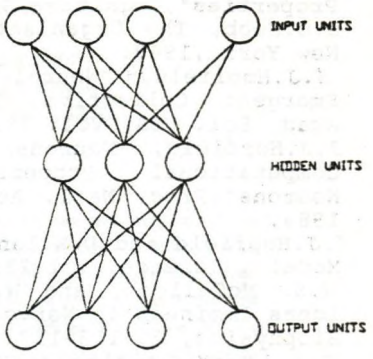


Fig. 3

The two main part of the system are NeurFrame Interpreter and the knowledge base for neural net informations. The three main elements of "conventional" (as shown in Fig. 1 and Fig. 3) neural nets are:

- the topology of the system
- the learning algorithms
- the information processing algorithms.

The system based on

- INTEL 80386 based computer (16 Mhz or more)
- MProlog version 2.3.
- MProlog E-Shell (MProlog Execution Shell)

The NeurFrame realises

- possibility to use "Query the user" technique
- how and why questions
- fast natural language interface
- trace techniques
- incremental system development

SUMMARY

The NeurFrame offers a good experimental system for modelling neural net architectures. At present it works a little slowly. To speed up in the future I would use the external language interfaces to get a faster arithmetics and for making a fast simulation for static parts. For more information about the system please contact the author.

REFERENCES

[1] J.A.Feldmann and D.H.Ballard, "Connectionist Models and Their Properties",Cognitive Science, Vol.6,205-254,1982.

[2] D.O.Hebb, The Organisation of Behavior, John Wiley & Sons, New York ,1949.

[3] J.J.Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities",Proc. Natl. Acad. Sci. USA, Vol. 79, 2554-2558, April 1982.

[4] J.J.Hopfield, "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons",Proc. Natl. Acad. Sci. USA, Vol.81, 3088-3092, may 1984.

[5] J.J.Hopfield and D.W.Tank, "Computing with Neural Circuits: A Model", Science, Vol 233, 625-633, August 1986.

[6] W.S. McCulloch, and W. Pitts, "A Logical Calculus of the Ideas Imminent in Nervous Activity." Bulletin of Mathematical Biophysics, 5. 115-133, 1943.

[7] C.A. Mead, Analog VLSI and Neural Systems, Course Notes, Computer Science Dept., California Institute of Technologie, 1986

[8] S.Grossberg, The Adaptive Brain I:Cognition, Learning, Reinforcement, and Rhythm, and The Adaptive Brain II: Vision, Speech, Language, and Motor Control, Elsevier/North-Holland, Amsterdam 1986.

[9] T.E.Posch, "Modells of the Generation and Processing of Signals by Nerve Cells: A Categorically Indexed Abridge Bibliography", USCEE Report 290, August 1968.

[10] R. Rosenblatt, Principles of Neurodynamics, New York, Spartan Books 1959.

[11] D.E Rummelharrt, G.E. Hinton, and R.J. Williams, "Learning Internal Representations by Error Propagation" in D.E Rummelhart & J.L. McClelland (Eds.) Parallel Distributed Processing: Explorations in the Microstucture of Cognition. Vol. 1: Foundations. MIT Press 1986.

[12] D.E Rummelhart & J.L. McClelland (Eds.) Parallel Distributed Processing: Explorations in the Microstucture of Cognition. Vol. 1: Foundations. MIT Press 1986.

[13] T. Sejnowski and C.R. Rosenberg, "NETtalk: A Parallel Network That Learns to Read Alound", John Hopkins Univ. Technical Report JHU/EECS-86/01, 1986.

[14] B. Widrow, and M.E. Hoff, "Adaptive Switching Circuits", 1960 IRE WESCON Conv. Record, Part 4, 96-104, August 1960.

Formality in Software Specifications

Attila Farkas & László Kiss

Advanced Systems Department
Central Research Institute for Physics
P.O. Box 49 Budapest 1525 HUNGARY

ABSTRACT In trying to cope with the ever growing size and complexity of applications, the software community seeks new ways to overcome the difficulties involved. Formality seems to be a promising paradigm for remedying the acute problems of the software process. The idea of formal software development is not new. Automatic Programming (AP), a classical branch of Artificial Intelligence (AI), has dealt with related problems for four decades. However, to make formal software development applicable for realistic applications, there is a need for further experience and researches in which AI and Software Engineering (SE) interweave. A crucial point of the software process is the method of specification, which is the main theme of this paper. In the first part, a review of the software process is given with an emphasis on specification related problems. It is followed by an overview of practical and experimental specification methods. Finally, a brief survey of formal specification languages is presented.

I. INTRODUCTION

The methods and tools in SE are in permanent advance, but more and more complex applications are undertaken, so the problem we call software crisis is not an issue of the past. Software is rarely produced on time or within budget, and when delivered it often fails to meet the users' needs.

Two major problems have been identified as being responsible for the majority of software project failures. These occur during requirements specification and maintenance, two highly iterative phases of the software life cycle.[1,2,3]

Errors committed during requirements specification are particularly dangerous as further steps of the development process (i.e. design and implementation) heavily depend on this stage. The later a specification bug is revealed the more the fixing can cost. A specification bug revealed during design or implementation phase may lead to redesigning or reimplementing significant parts of the system, or it can even cause the failure of the whole project.

The life of a system does not end with delivery, possibly a number of modification needs arise when the system is in use. The major source of problems in this phase, called maintenance, seems to be that modifications are performed on the source code. It is tough going even if proper specification and documentation of the system exists (i.e. agreeing with the code), since the correspondence between these documents and the optimized source code can hardly be traced.

Due to the increasing number and size of software projects, the above problems are even more distinct today. The software development process is becoming manually unmanageable: there is an urgent need for automation and

automatic aids. Besides making the traditional approach in SE (i.e. the waterfall model and related methods) more perfect, the software community seeks new paradigms (e.g., rapid prototyping, operational approach) for the software process. The quest for automation has led to research in the areas of development methodologies, specification and implementation languages, and automatic management aids. As a result of this research, more formal approaches emerged in which the specification has gained a more prominent role.

Creating a specification is by no means a trivial task, and is particularly less so when specifications are becoming more formal. The creation of a specification involves understanding the problem at hand and rephrasing the relevant knowledge in the adopted specification language. A good specification language should facilitate both of these activities.

This raises three concerns about specifications. First, what should a good specification contain? Second, what is a good specification language like? Third, what kind of techniques should be used to create a specification?

This paper, giving a survey of specification languages and methods, examines how ideas from mathematics, SE and AI influenced the answers to these questions. In the next section a brief overview of software development approaches is given. In Section III specification methods and techniques are discussed. Section IV presents a brief overview of formal specification languages, and finally a conclusion is drawn on the directions of future developments.

II. THE SOFTWARE PROCESS

The objective of this section is to illustrate how formal methods are coming into prominence in SE as a promising paradigm for remedying the acute problems of SE. In the following, classical and new SE problems, approaches and perspectives are discussed.

The software crisis. The object of software engineering is the study and evaluation of techniques that efficiently and cost-effectively produce reliable software products to satisfy the users' needs. While we have good methods and tools for programming in the small, there are major difficulties in developing large-scale software systems. Software is rarely produced on time or within budget, and when delivered it often fails to meet the users' needs. Another problem is that with the increasing number of applications, the maintenance of existing softwares lays an ever growing burden on software companies. The existing software development tools and methods can not cope with the new applications of ever growing size and complexity.

The most widely used, and most well-understood general method for system development in SE is the waterfall model. According to this model, the software life cycle is divided into six phases. These are: requirement understanding, requirement specification, design, coding, testing, and maintenance.

Two main problem areas have been identified as being responsible for most part of software project costs. These are usually referred to as the requirements problem and the maintenance problem.[1,2] In the rest of this section these problems are characterized.

The requirements problem. The first task during the software development process is understanding the current users environment, its deficiencies, and the requirements imposed on the software system to be built. Acquiring all the necessary information from the users is not a trivial problem. The difficulties partly lie in the different professional backgrounds of the computer experts carrying out the requirements analysis and the users of the current system. Users are possibly not skilled in computer sciences and system analysts do not know the application domain. Misunderstandings occurring frequently during requirements specification originate mainly from this communication bottleneck. Furthermore, the great amount of information handled during requirements specification needs some means of managing. As faults committed in this early phase of the software development process are very costly considering both time and money, elimination of these problems has become a very intensive research area.

Advances in the conventional approach. Research striving for improving the conventional approach first made it clear that it is worth spending more time with the specification of requirements and start coding only after ensuring that the specification reflects the right requirements. Requirements specification has become a highly iterative process in which feedback from end users is of high importance. To improve communication, which proceeded mostly in natural language in early times, new means of communication have also been developed. Though natural language communication has its obvious advantages, for example, it is the most convenient way of stating requirements and producing the documentations for the users, this approach has its own drawbacks.[41] Natural language texts are subject to contain ambiguous and contradictory statements, which are hard to recognize. Users are reluctant to read big three ring binders, even if the text is well structured, so the documentations specifying the proposed system are often poorly criticized. In trying to improve the communication facilities, methodologies provide more and more automated tools and techniques supporting problem understanding and problem solving.

The evolution of the waterfall model has led to methodologies in which a larger amount of time is spent with developing specifications and the validation of the specification has a greater emphasis. Experiences have shown that these amendments, though beneficial, have not solved the communication problem satisfactorily.

A fundamental and inherent problem undermining the conventional approach is that it uses only "static" means of communication (e.g., natural language documents, diagrams). Although the various documents comprising the specification may be appropriate for defining the proposed system, validation of the system relying on merely paper-based descriptions is often insufficient. The user cannot gain deep understanding of the system's operation. Some way should be found to support the "visualization" of the desired operation. Furthermore, specification checking facilities provided by this kind of "semi-formalized" techniques are of limited scope. CASE (Computer Aided SE) tools are also of limited skill due to the inherent defi-

ciencies of the manual techniques they automate.

Alternative models. Alternative approaches conveying new paradigms have been developed for overcoming the drawbacks of the conventional model. The conventional model and the proposed alternatives attack different aspects of the original requirements problem. Namely, the conventional approach lays stress upon requirements specification to reduce the number of specification bugs turning up in the time consuming and costly coding phase (or even later in the life cycle). Several alternative approaches try to reduce the cost of possible reimplementations by facilitating coding in various ways, while others try to test (e.g. validate and verify) the specification rather than the code. The most common alternatives to the conventional life cycle model are the following:

Very high level language (VHLL) approach. A very natural approach to automating the development is to continue the historical evolutionary trend in developing ever more powerful and expressive programming languages; this is the very high level programming language approach.[4] The advantages achieved by programming in a VHLL are typically the implicit specification of control flow and the high level data referencing. Programmers using a VHLL can write programs with few errors. However, VHLLs are difficult to use if the application domain is not well understood.

Application generators (AGs). Another approach to improve software productivity is to use application generators.[5] AGs support primarily data processing. They possess all the "knowledge" about a special problem class, and therefore can be "programmed" in a very high level, special purpose language in order to quickly generate a program. A well-known example is the IBM's RPG. AGs can be viewed as very simple forerunners of knowledge based automatic programming systems.

Operational approach. The operational approach is an alternative paradigm for implementing systems[6,7]. In this approach, in contrast with the conventional development, the specification is executable. It describes the behaviour of the target system and it can be experimented with. The behaviour of that system may be modified by altering the specification. When the specification has been accepted as valid, an efficient implementation can be produced[8].

Rapid prototyping. The operational approach is strongly connected with the idea of rapid prototyping, since the executable specification serves as an initial version of the target system[9,10]. Rapid prototyping can also be used as a tool in the conventional approach to make the users' demands clear.

Many of the alternative approaches enumerated above are still in an experimental state, lacking the maturity of the conventional approach. None of them are a full-fledged method spanning the whole software process, and they lack integrated automated aids commonly available for conventional methodologies.

The maintenance problem. Though the development of the above approaches eased the requirements problem to some degree, it did virtually nothing for eliminating the equally serious maintenance problem. These approaches provide only "implicit" help: a well-designed, well-implemented system with good documentation is easier to maintain. However, these improve-

ments still seem to be insufficient. The maintenance phase, which starts when the software product has been delivered for use, usually constitutes more than 60 percent of the life cycle cost.[1,2,11]

Maintenance has two basic facets: tracking down and removal of bugs that show up, and modifications to meet environmental changes. Though the two kinds of maintenance activities may diverge in details, they can be decomposed similarly on the top level. A maintenance event can be divided into three successive though not entirely distinct parts:

- First the problem necessitating maintenance needs to be understood. The cause of a problem may be a software malfunction caused by either a misunderstood requirement or a bug in the source code, or some changes in the users environment/requirements.

- The next step is to identify those highest level parts in the documentation hierarchy (including the specification, different level designs, and the source code) which are still affected by the particular maintenance activity. It may be the case that modifications affect the source code only (bug elimination), the system design as well (software/hardware changes), or even the specification (requiremental changes).

- After introducing the required modifications at the highest level, changes must be propagated through the lower levels until a new implementation, agreeing with the documentation and satisfying the maintenance demands, is obtained.

Although maintenance is not free from the type of errors discussed at the requirements problem, they are less characteristic here than at the specification phase where the problem of understanding requirements dominates. The laborious nature of maintenance derives from the need to understand and modify already existing large, coherent structures. What makes maintenance difficult, and different from development-time modifications, is that modifications must always be propagated as far as the level of code.

Navigating through the documentation set and finding the relevant parts are impeded by the common situation that, despite the fairly good specification, design, and implementation tools and techniques, the dependencies among these different level descriptions are vague and poorly documented. The problem is that there is no presently available technology for managing knowledge-intensive, informal processes like software development. As a result, information about the development process and the rationale behind decisions is mostly unavailable. It is a particularly sore problem when we have to perform modifications on the source code. During coding and tuning, much labour is expended on optimizing the implementation. As a side effect of these optimizations, the relationship among the source code, the design and the specification becomes largely obscured. It is hard to track down related code chunks without knowledge about optimization decisions, for optimization often scatters logically connected parts of the code. These deficiencies make the software hard to understand for the maintenance team which, in addition, frequently consists of completely new people in the system's life.

One possibility for reducing the costs of maintenance is reducing the number

of maintenance events. Methods emphasizing early operation such as rapid prototyping and executable specifications support early bug detection, thereby decreasing the number of bugs emerging in the maintenance phase. Well-built systems decrease the costs of transplantations to new software/hardware environments, so approaches supporting the development of portable systems (e.g. application generators, reusability) alleviate this problem to some extent. A prudent design can take into account the foreseeable requiremental/environmental changes.

A promising way of giving a more perfect and overall solution to the maintenance problem is to increase the level at which maintenance is performed. In [2] the authors argue that the satisfactory solution would be to maintain directly the specification and reimplement the system with automated support. However, it demands a new, automation based paradigm for software development.

Automation based software development. The necessary degree of reduction in software development costs can hardly be expected of enhancing the manual process, some kind of automated support is needed. The most ambitious endeavours aim at full automation, that is, code generation from the specification without human assistance. However, the approaches picturing the development of the implementation as an interactive process are more feasible for the near future, for automatically generating efficient code from high-level specifications is far more difficult than compiling high-level programming language code.[2] A large number of alternatives have to be considered at each development step on the way from specification to code, and, due to the wide gap between high-level specifications and implementations, it is very hard to tell how each alternative affects the global properties (e.g., efficiency) of the final product.

According to the interactive development approach, the programmer makes all the critical high level decisions by selecting from implementation options offered to him by the code generator at the proper time.[2] Making use of this information, the code generator gradually transforms the initial high-level specification, via a series of intermediate level specifications, into a low-level specification which can be automatically compiled into efficient code.

According to this approach, during maintenance, any modification is introduced first at specification level, and then the modified specification is refined down to a new design and implementation. In this way, performing maintenance directly on the specification means that the amount of work needed to propagate specification changes through design and implementation is significantly cut.

III. SPECIFICATION METHODS

As it appears from the previous section's overview of the software process, requirements specification has a distinctive role in software development. The final quality of the specification (e.g., precision, consistency, relevance) and the overall work required to develop it highly depends on the method by which the specification is constructed. This section surveys specification methods and techniques currently used and experimented with.

Specification in practice. Practical methods are essentially based on the support of human problem solving and problem understanding processes. In order to improve the efficiency of the program development process we should first understand the mechanisms inherent to human cognitive activities. The research done in this area tends to interweave with AI research. In seeking better specification methods, specification methods applied in current practice serve as a natural starting point.

Practical specification methods involve natural language communication supported with formal elements where these are naturally applicable. The specification method is often interactive - checking hypotheses, pointing out inconsistencies, and asking the user for further information.

The analysts armed with their skills, various techniques and automated aids and guided by the choreography prescribed by the adopted development methodology produce an informal description of the required system. This description serves as the system specification for the designers/programmers. The techniques and support tools used in the requirements specification process are based on common human "stunts" of thinking.

During specification, it is a commonly used technique to help explanation with some drawings. Both users and analysts show a preference for describing structural (organizational) information with various kinds of diagrams. The use of diagrams indicates a kind of "natural" need for some formality to help explanation and understanding. To satisfy the need for such "thinking aids" and to provide the development team with a unified means of communication, practical methodologies now offer several standard diagramming techniques (data flow, control flow, structure chart, etc.) together with instructions for their use. However, these diagrams often have loose semantics, as they can be used in several ways to describe the same information. Another problem is that the various diagrams in a specification are hard to cross-check by lack of an underlying framework connecting them. Experiences with CASE tools have also identified the lack of powerful formal models from behind these tools as the obstacles of providing more substantial computer assistance.

Advantages of increasing the formality in specifications can be summarized as follows:

- Rephrasing parts of the natural language specification in a formal language allows establishing consistency criteria to detect internal inconsistencies in the acquired information.

- The structured, formalized methods analyse the system from several points of view (e.g., flow of data, control structure, etc.). The redundancy introduced by multiple views helps ensure the completeness of information. Besides, the comparison of information captured by the different views provides additional consistency criteria.

- Formalization of the information gathered about the system is a kind of problem solving activity, during which a number of questions are drafted in the system analyst's mind. As difficulties in specifying the requirements derive from the fact that users can not differentiate between information relevant and irrelevant for analysts, it is vital that the questions of the analysts govern the user interviews properly.

- Applying structured and formalized methods usually means that considerable parts of the gathered information is expressed in the form of diagrams. The diagrams used in methodologies tend to be of simple structure in order to be intelligible for users too. Experiences have shown that the willingness of users to participate in the tiresome validation process significantly grows when the material to be validated is given in some graphical form.

- Formalization of the specification allows the application of computers from very early phases of the life cycle, e.g. requirements analysis and specification. It gives rise to writing programs that check the requirements for completeness and consistency or support the drawing of diagrams involved in specifications.

Taking a rather technical and managerial point of view, it is important to represent specifications in some exact way for several reasons:

- For the storage capacity of a human's memory is limited, a description is needed which presents the information in a form easy to comprehend and manage.

- Members of the development team need to rely on each-other's work. Thus the specification must be a description of the "digested" information, i.e., containing only the relevant information organized in a way easy to understand.

- Regardless of the extent of the formality, a specification may be regarded as the basis of a "contractual agreement" between the two parties who want the system built and the parties who will actually do the building.

Specification for formal development. Formal software development is a process during which a formal description (e.g., one with precise syntax and semantics) of the users requirements is produced, and then starting from this formal description, a target language program is constructed by some formal method (e.g., theorem-proving, gradual transformation). The process of constructing the program may be manual, fully automated, or partially automated allowing high level decisions to be made interactively. The process of constructing the formal specification may also proceed "manually", when the analyst directly puts down the requirements in some formal language, or with computer assistance, automatically converting informal to formal.

Formal specification methods. Formal specification methods are those requiring the analyst to rephrase informally stated requirements in a language with precise syntax and semantics. Depending on how the development proceeds from the specification, these formal languages can be grouped into two classes.

When the theorem proving approach is adopted for the development, the specification language is some logic-based language. Here a program is specified as

$$\forall x \, (P(x) \dashrightarrow \exists y \, Q(x,y))$$

which states that for all values of input variables to the program, x, for which the predicate P(x) is true, there are output variables, y, such that Q(x,y) is true. This expression would then be given to a theorem prover that produces a proof of the statement from which a program can be extracted as a side effect.[12]

When the more promising transformational approach is used for program development, the specification is written in some so called very high level language. These languages encourage the use of entities that are not immediately implementable on a computer, or at least not implementable with some desired degree of efficiency. The program then is developed from the formal specification via a series of small refinement steps, inspired by the programming discipline of stepwise refinement:

$$S_0 \text{ ---> } S_1 \text{ ---> } \ldots \text{ ---> } S_n \text{ ---> } P$$

where S_0 is the formal specification, P is the program written in the target language and S_i (i = 1, ..., n) are intermediate level descriptions. Each refinement step (S_i ---> S_{i+1}) captures a single design decision. If each individual refinement step can be proved correct, then the program P itself is guaranteed to be correct.

The application of formal specification languages also needs tools and techniques commonly used in practical specification methods. Though specification in a formal language has its own advantages, it also raises new problems not typical of informal techniques. Namely, informal techniques lack the rigour of programming languages, whereas formal languages lack the ease of expression permitted by informality. Research has been directed towards combining the advantageous features of both approaches.

On the one hand, formal specification languages with ever more convenient syntactic and semantic constructs are being developed. A more thorough discussion of these languages is presented in Section IV. Other research, mainly in the field of Expert Systems (ES) applications, aim at automating the process of converting informally stated requirements into a formal description. Next, to illustrate this research, several experimental systems are described. These systems utilize various techniques commonly used by analysts during the requirements specification phase.

Natural language specification. Natural language is the basic means of communicating requirements. As requirements come from the user mostly in this form, any specification method has natural language as its base, regardless of the form in which the acquired information is finally represented. A significant part of the analysts' job is the translation of the natural language requirements into a formal representation. A seducing way to make specification easier is to entrust this job to an automated system.

That is why intensive research has been pursued with natural language specifications. A number of experimental systems have been developed in this area.[12] Generally speaking, these systems take some natural language text as input and, mostly after an interactive dialogue, have as their output a formal description of a program to be written. Two classical systems are the NLPQ[12] and the SAFE[12] systems.

NLPQ was the first AP system to utilize natural language dialogue as a specification method. During an English dialogue on a simple queuing problem the system gradually builds a semantic net as the formal specification of the problem. The system produces questions to acquire missing information and answers questions about the state of the model being built. Starting from the completed semantic net the transfor-

mation module of the system then produces a GPSS program.

The SAFE system, a module of an AP system spanning the full program development, accepts program specifications given in the form of preparsed English sentences with limited syntax and vocabulary. To eliminate ambiguities and complete the initial specification, the system leads a dialogue with the user. The output of the system is a complete and consistent specification of the problem written in a high level language called Gist.

Specification by examples. It is an often useful and convenient way of demonstrating the desired behaviour of a program to simply give examples of what the desired program is to do.[12,13] This kind of specification might consist of examples of the input/output behaviour of the desired program, or it might consist of traces of how the program processed the input. This method is often used for specifying simple input/output programs, and it is a comfortable way for laymen to explain simple algorithms. There are many difficulties involved in specification by examples (or traces); for instance, this kind of specification is rarely complete, since a few examples will not fully describe the behaviour of the desired program in all cases. Another problem is its heuristic nature and strong dependence on domain and everyday knowledge. Experimental systems which try to catch the mechanisms behind this kind of specification technique provide models which are completed by giving examples to the AP system.

A well-known system utilizing the specification by examples technique is the PSI knowledge based AP system.[12,14] The PSI system was intended to be used as a programming assistant in the field of symbolic programming. A subsystem of PSI allows specifying programs by giving examples of input/output pairs and program traces. There is also a natural language dialogue facility provided by another subsystem. Besides which, the system gives direct access to the formal model, called program net, which is being built on the base of the natural language dialogue and the given examples.

Critiquing specifications. An old and tried technique used in requirements analysis is generating examples for extreme cases to test completeness and to perform some kind of validation. An experimental system imitating this trick of system analysts has been built by Fickas et al.[15] This system has a model about the application domain with issues which frequently come up both in general and in the application domain about software systems. Attaching importance weights to these issues, the system evaluates the extent to which the specification supports the goals the weighted issues indicate.

IV. FORMAL SPECIFICATION LANGUAGES

Creating a formal specification involves understanding the problem and its requirements and rephrasing the acquired information in a formal language (the specification language). Writing formal descriptions of application specific knowledge is not an easy task. The problem of formalizing knowledge frequently arises in the area of AI applications. The field of knowledge representation, one of the most important branches of AI research, concentrates on this problem.[37,42] It

aims at developing knowledge representation techniques which facilitate describing domain specific knowledge and allow flexible manipulation of the represented information. The difficult nature of formal description is well illustrated by the common AP experience that very often the main problem is not how to generate code from the specification but how to specify the problem in terms of formal constructs. Formally specifying the problem for an AP system can often be as laborious as manual coding, whereas the same problem could be explained to a human programmer in a few words. A characteristic feature of the specification should be, however, that this kind of representation is orders of magnitude easier to develop than actually implementing the system.

One of the key conclusions of the past two decades of knowledge representation research is that the efficiency of information passing during man-machine interaction can only approximate that of human-human communication if the machine is already equipped with a great deal of knowledge about the application domain.[16] Regarding specifications, this means that effective and convenient specification requires the use of specification languages well matching the characteristic features of the application area. This section surveys specification languages according to an application domain based classification.

Application areas can possibly be categorized by several points of view and with different levels of detail. For the purposes of this paper, the basic principle of categorization has been to separate application domains according to how differently they "behave" on a superficial level when it comes to describing them. That is, categorization is based on the different characteristic features of specification languages suitable for describing the various problem domains.

A computer system can be viewed as a model of a "world" or "slice of reality" about which it contains information.[17] According to this view, software specification is actually a model building activity. This activity is considerably different, however, in cases when the "world" to be modeled is some part of the real world compared to those cases when this "world" exists only in a human's mind. This is our first separating principle for categorizing application domains.

When some part of the real world is to be modelled, it is a common problem that nobody can exactly tell how far the model should extend, which aspects of the world should be modelled and which should be left out. Clearing up ambiguities and defining the system is a time consuming process which proceeds through a series of user interviews. This process is subject to errors discussed at the requirements problem in Section II. Specification languages should facilitate communication between analysts and users by being readable for users too. They should also support top-down decomposition and specification of the problem so that they can be used in early stages of the development process when understanding of the problem is rather superficial.

Another important aspect of the real world is that its objects "tangibly" exist, that is, they exist uniquely and cannot be multiplied. According to a somewhat simplified view, people map real world objects and concepts onto some conceptual units associated with

the relevant information, and think in terms of these abstractions. In order to match the human way of thinking, specification languages should facilitate this "one-to-one" correspondence. Another characteristic of the real world is that relations between, and operations on, its objects are relatively simple, partly due to special properties of real objects discussed above, partly due to the limits of human mental capacities (most applications automate some manually performed job).

On the other hand, modelling is impeded by the inherent property of the real world that practically all rules have exceptions. Irregularities arising frequently in reality cause many problems when we try to force some portion of the world into a formal and simplified model. These exceptional cases usually contribute to the complexity of the model on a large scale. Specification languages should directly support handling degenerate and exceptional cases in order to avoid becoming unmanageable models.

Within the area of real world applications two major subareas can be defined according to whether the static or dynamic aspects of the world dominate in the model. These are the areas of information systems and real-time systems, respectively. Information systems deal primarily with the structural and organizational issues of the world. They store and maintain information about relevant real world objects and the state of relations between them. Information systems are concerned with real world events only so much as they change the state of the modelled portion of the world. On the other hand, real-time systems (often called embedded systems) concentrate on the communication among the objects of the world. They represent real world objects in a direct way and so to say simulate the communication among objects.

When the world to be modelled is an imaginary one, the communication problem is not characteristic. A significant part of the modeling job need not be carried out in such cases since the "world" to be modelled is often given in the form of an abstract model. A further advantage is that models of this kind are usually fairly "regular", constraints on objects and relationships are generally satisfied "by definition". On the other side, in an artificial world there is practically no limit for the complexity of the relationships between and the operations performed on objects. To cope with this indefinite complexity the most generic constructs of mathematics are required, such as sets, logic assertions, functions, etc.

The general purpose specification languages, some of which are discussed later in this section, provide such general constructs as the basis of description. Application domains for which these general purpose specification languages often prove appropriate include, for example, support systems (operating systems, editors, etc.) and mathematical software. Of course, it is possible to divide these large application areas into smaller ones for which more specific specification facilities can be developed, as, for example, languages for specifying editors actually exist, discussion of this vast area is far beyond the scope of this paper.

In the rest of the paper, a brief survey of specification languages is given. Some general purpose specification languages are presented first, followed by a discussion of specification languages for information systems and

real-time systems. Since the borders of the categories may not be drawn sharply, and since some of the languages enumerated are intentionally developed for use in several domains, the classification of specification languages below is not unanimous.

General purpose specification languages. As general purpose specification languages are designed to be suitable for a very wide range of application domains, there are no common properties which all of these languages share, or should share (except for obvious features like ease of expression, readability, etc.). However, they can be classified according to the underlying mathematical theories, that is, the set of mathematical instruments, they use as a basis of expression. In the following, some of the prominent general purpose specification languages will be briefly described and grouped by their underlying approach for modeling.

Model-oriented languages. One way to specify software is to describe the mathematical models of the structures and operations on those structures one intends to represent. Languages taking this model-theoretic approach to modelling are called model-oriented languages. Typically, the matematical models of the structures are abstract data types, and employ the ideas of sets, functions, relations, sequences, Cartesian products, etc. Operations are also specified that are required on the abstract model. Pre- and postconditions are used to specify procedures, which may have side effects. A problem with model-oriented specifications is that it is easy to overspecify a system, eliminating certain implementations from consideration from the beginning.

Two prominent representatives of this category are the Z language[18,19] and the specification language of the Vienna Development Method (VDM).[20,21]

The Z specification language is a formal notation for ordinary naive set theory. It is based on the principle that programs and data can be described using set theory just as all of mathematics can be built on a set-theoretic basis. Data types are modelled in Z using set-theoretic constructions just as natural numbers, ordered pairs and sequences are defined in set-theoretic terms in mathematics.

VDM is a mathematically based set of techniques for the specification, development, and proof of correctness of computer systems. Its objective is to retain most of the advantages of formal program development without the foundational overhead of formality. VDM adopts the transformational approach, programs are produced by a process where the individual refinement steps are shown to be correct using arguments which are formalizable rather than formal, thus approximating the level of rigour used in mathematics.

Rewrite rule based languages. During the past decade a number of very high-level programming languages which can be seen as executable specification languages have been developed. The underlying idea here is that equations can be used as rewrite rules. That is, an equation $\forall x(t = t')$ can be viewed as a rewrite rule t ==> t' (or t' ==> t) saying that any substitution instance of t in an expression can be replaced by the corresponding substitution instance of t'. Under certain conditions, it is possible to "run" a set of such rules to compute the value of an expression. Well-known examples of rewrite rule based languages are HOPE,[22] Standard ML[23] and OBJ2.[24]

Algebraic specification languages. Algebraic specification languages, such as CLEAR,[25,26] ASL[27] or Extended ML[28], are those modelling programs as many-sorted algebras. The idea behind this specification method is that a functional program can be viewed as a many-sorted algebra, i.e. a number of sets of data values together with a number of total functions defined on the sets. Each data type of the program is modelled by one corresponding set of values. The functions defined on these value sets correspond to functions in the program. A specification consists of a signature (a set of data type names and a set of function names with their types) together with a set of equational axioms expressing constraints which the functions must satisfy.

This approach allows focusing on the representation of data and the input/output behaviour of functions by abstracting away from the algorithms used to compute functions and their implemenation on a given programming language. It is possible to extend this paradigm to handle imperative programs, too.[29]

Specifying information systems. By now information system (IS) applications have grown into one of the largest subarea of SE. The great number of commercial applications bearing similar characteristics and problems have urged research toward specification languages which better exploit these similarities. Tools and techniques for developing the specification have also gone through a dynamic progress. Several generations of programming and specification languages have succeeded each other in the past decade obeying the aphorism "Today's specification languages are tomorrow's programming languages". The relational data base theory, which was originally intended to be a specification tool, has become one of the most popular organizational principle of today's DBMSs. The next stage of IS modeling is marked by the appearance of Chen's Entity-Relationship Model[30] which offers a better data abstraction facility. Many of the current IS specification languages are based on this or similar semantic data models.[31] However, data is only one aspect of ISs to be dealt with. In the following, we briefly survey current specification languages which intend to provide a coherent framework for describing both the static and behavioural aspects of the world.

Conceptual modelling languages. Languages traditionally used for IS development lay the emphasis primarily on the data to be stored, and usually are based on some data modeling technique (e.g., relational). The fundamental problem with these languages is that the primitives they offer (records of numbers and strings, relations, etc.) are more appropriate for specifying how data is stored and accessed in the computer than modelling the underlying concepts. According to a somewhat naive view, people perceive and think about the world in terms of basic concepts which correspond to objects of the real world in some way. The knowledge associated with a real word object is organized around the corresponding concept. An IS is actually a conceptual model which operates on such basic concepts, describing a portion of real world interesting for the user. It usually stores, retrieves manipulates information about the relevant portion of the real world, and in an important sense can be viewed as model of that world, or more accurately of the user's conception of the world. The utility of

an IS is that the user can answer questions about the world more quickly and easily by inspecting the IS (i.e., the model of the world) than by actually performing "measurements" in the real world.

The so called conceptual modeling languages (CML) recently developed for eliminating the deficiencies of traditional IS development are rooted in this model oriented view. Languages fitting into this line, like TAXIS[32], Galileo[33], Adaplex[34], Dial[35], diverge in some aspects, such as syntax, scope, ease of usage, development stage, etc. but can be tightly related on the basis of their characteristic features, and the way they approach the specification of IS requirements. Describing these languages is beyond the scope of this paper, we rather discuss a blend of their features and deficiencies in the following.

CMLs claim to allow more natural and direct modeling of the world than has been the case traditionally, and conceptual models are claimed to capture the semantics of a situation more accurately and conveniently.[36] CMLs have been used at the requirements, design or implementation level for ISs. They have a number of features distinguishing them from traditional approaches.

- First of all, they allow a natural, one-to-one correspondence between the relevant entities of the world and the objects in the model. This makes modelling easier by reducing the job to a natural, direct mapping of the analyst's conceptualization of the world.

- Objects of the model can have various properties/attributes. In most CMLs, they are allowed to have multiple values. Properties can also be defined whose values are computed, rather than explicitly stored.

- Objects are grouped into classes for the purpose of capturing common characteristics. Every object is required to be an instance of at least one class. This can be viewed as a kind of typing mechanism, for class descriptions impose constraints on attributes applicable to instances and their values.

- Classes are organized into subclass hierarchies. A new subclass can be created by introducing additional detail to the description of an existing one. New superclasses can be composed by abstracting out some common properties of several existing classes. Class hierarchies are usually combined with some kind of inheritance mechanism to eliminate unnecessary duplication of information in subclass definitions. By inheritance, subclasses automatically have the properties defined for their superclasses.

- Events, which are entities having an associated time of occurrence or initiation/termination, can be described as objects. Several CMLs provide semantic relations like "must be preceded by" or "must be followed by" between event type objects, allowing thus one to define constraints on event sequences.

The current CMLs have not yet resolved all the problems of IS development. They still lack important features for conveniently specifying ISs. While they provide powerful facilities for data abstraction, they often prove to be weak at describing the dynamic aspects of ISs. CMLs usually do not allow for specifying temporal aspects of the world in sufficient detail and on a convenient level. Another deficiency is

that these languages provide practically no means for attaching nonfunctional requirements to the specification.

Considering a more practical issue, the process of building the specification is poorly supported. To use these specification languages for problems of realistic size, more appropriate information acquisition facilities should be provided. For example, they do not provide constructs for specifying partial conceptual views of the system. This limits the use of these languages in the early stages of the development when understanding of the problem is poor and is made up of knowledge fragments each of which captures only certain aspects of the system.

Requirements modelling languages. Requirements specification in current practice is often restricted to functional aspects, prescribing only the tasks to be carried out by the future system. During requirements specification, system analysts attain a lot of additional information for it is inevitable that they get acquainted with the environment in which the system will operate. It is argued that this knowledge should be captured in the form of a model of the users environment and the various requirements should be stated in relation to this model.[37]

There would be several uses of such a model. To mention only the most salient ones, it could significantly imp rove the user-analyst communication by providing the most natural context for users and giving a base for automated aids; furthermore, being in a form processable by the computer, the model can be used to simulate the world offering dynamic checking facilities.

Requirements modeling shows similarities to semantic data modeling. A significant difference, however, is that requirements modeling captures a broader context of environment. For example, a requirement model may contain concepts which will not be implemented in the computerized system (e.g., people or other computer systems communicating with the system). Another difference is the approach taken for handling the dynamic aspects of the world. A requirement model is not made up of snapshots of the world in different moments of time, rather it depicts the objects of the world as "three-dimensional" snapshots, incorporating a time concept, similar to that of a human, in the model (temporal properties, relations, etc.).

Based on this idea, an experimental language is being developed by Greenspan et al.[38] This requirements modeling language, called RML, was developed to serve as a specification language for the TAXIS[32] programming language. RML shares the underlying principles of TAXIS (e.g., uniform treatment of data and transactions, organizational hierarchies of objects, etc.) and the basic model building blocks, such as objects, classes, and properties.

In a coherent object-oriented framework, RML integrates three different viewpoints of the world being modeled. Objects can be of three general types, each for describing one of the three views: entities can be used for describing domain objects, activities for defining events in the world, and assertions for imposing constraints on the world.

The main advance in RML compared to TAXIS (and other CMLs) is the built in

time concept which allows one to explicitly state constraints on temporal relationships. The notion of time in RML is based on an infinite and dense sequence of time points. The question whether an RML object exists or not is meaningful only with respect to a specified time point or time interval. The attributes of an object as well as its membership in a class is always evaluated for a given time. Assertions are also affected by time: they can have different values (true or false) at different times.

Due to the powerful constructs and facilities provided for organizing and abstracting details, RML allows convenient specification, and at the same time it retains the advantages of formality since it has a formal semantics definition in a logic with time.[38]

Specification of real time systems. Another large class of computer applications is that of real-time systems. This field becomes particularly important today, when the increasing demands for industrial automation require more and more complex software systems for production and industrial robot control. Real-time applications are also gaining ground in other areas, such as flight guidance systems, patient monitoring systems, radar tracking systems, and data acquisition systems for experimental equipment.

The most common feature characteristic to all real-time applications is process control, that is the provision of continual feedback to an unintelligent environment. This continual communication with an unintelligent environment imposes strong performance constraints (e.g., reliability, real-time response requirements) on these systems. It seems that this emphasis on performance requirements is what really characterizes real-time systems, and cause to be more aware of their roles in their environments than we are for other types of systems.

The high complexity of the interface between the system and its environment is typical in this domain. The interfaces are usually asynchronous, parallel, and distributed, which makes the related requirements difficult to specify in a way that is both precise and comprehensible.

Another characteristic of the domain is that real-time systems can be extraordinarily hard to test. It is partly because of the complex interface, and partly because embedded systems often cannot be tested in their operational environment. Finally, these problems exacerbate the maintenance problem, too. Due to the difficulty of testing, it is vital that errors be detected as early as possible in the development process.

These characteristics have strong effects on the desirable features of specification languages. Since the interface is very complex between an embedded system and its environment, explicit modeling of the environment is desirable. Also because of the interface's complexity and because of the hard testability, it is desirable that specification be operational. The principle that a specification be free of implementation concerns is particularly important here. As real-time systems have sharp timing and space constraints to be fulfilled, it is very important that hidden implementation details do not exclude any possible solution.

Requirements definition methodologies used in this field tend to be ad hoc.[39] Unfortunately, a common characteris-

tic is that less is done in the area of specification languages tailored to this field than for ISs. It is likely to be due to the fact that the field is more diverging than that of ISs and also because such applications are characteristically harder to design and implement than to specify.

As implementation constitutes the major part of real-time system development, the most important concern in this field is to develop special purpose, high level programming languages. However, it is hard to achieve a generally applicable high level programming language in this field, since programming often inevitably involves assembly (or even lower level) coding. Due to the wide gap between specification and implementation languages in this field, specification languages tend to be operational. Simulation of the future system operation based on the specification can be beneficial considering that testable versions of the system are usually available only at very late stages of the development.

One of the most prominent and influential contribution to the development of real-time systems has been made by Zave et al.[8] They developed an operational approach (see Section II) for specifying real-time systems which is based on an executable specification language, PIASLey, with a transformational development method. The PAISLey language is an "almost" purely applicative language, including only features which are directly associated with run-time semantics. It has a decomposition facility based on processes which are the primary units of specification. In specifying a system with PAISLey, a system is first decomposed into processes, then computations making up processes are decomposed into steps. Applicative programming starts after this organizational level decomposition has completed, thus applicative programs have relatively small size. The asynchronous communication between processes constituting the system are modelled by three primitive functions coherently integrated into the applicative framework.

Executability is achieved by an interactive interpreter which asks for missing information from the user to interpret incomplete specifications. This kind of executability approach allows executing specifications regardless how incomplete and high level they are. Actually the specification under interpretation acts as a simulation of the system's behaviour.

A similar approach is taken by Goedicke in a formal specification language, called EDE.[40]

V. CONCLUSION

In trying to cope with the ever growing size and complexity of applications, attention is being turned towards formal and automation-based software development paradigms (as illustrated in Section II). By now it is clear that real alleviation of the software crisis can only be expected from shifting the man-machine participation rate in the software process towards the machine part, that is, for more and more automation of the development process. Independently of the degree of automation, however, AP systems will always require some input: the complete and unambiguous description of the problem in question and its requirements. As design and implementation are getting easier due to computerized assistance and high level implementation languages, the specification of re-

quirements performed with human participation tends to become the bottleneck of the software development process.

Current research into specification related problems is constituted of several research directions each being concerned only with certain aspects of the overall problem. This paper has tried to give a survey of these experimental research areas.

Some research attacks the specification problem along with other problems of the software process, from a classical SE point of view: seeks better methodologies for engineering software. The suggested alternative ways for developing software have been overviewed in Section II.

Methodologies for developing the specification have been discussed in Section III. Research in this field studies the procedure of building the specification and the tools and techniques usefully applicable in that activity.

A fundamental question considering specifications is the form of representation. The tendency here is the increasing formality to meet automation demands. Formality, on one hand, has ever had its own advantages and disadvantages. What makes formality worth advocating nowadays is the increasing demands for automation and the real chance of accomplishing it. A survey of formal specification languages has been given in Section IV.

The development of current specification methods and languages has been influenced by ideas mainly from three areas, SE, AI, and mathematics.

For example, general purpose formal specification languages incorporating ideas from mathematics and programming languages allow precise syntax and semantics definitions, and thus lend themselves to computer processing. Programming language and general mathematical constructs (e.g., sets, sequences, relations) have enriched specification languages with general means of formulating information rigourously but still conveniently.

The development of Conceptual Modelling Languages has been primarily influenced by ideas from AI Knowledge Representation. The object-centered framework these languages incorporate allows convenient modelling of the relevant concepts, and, by using generalization/specialization hierarchies of objects, requirements specification can be performed in a top-down fashion. This top-down decomposition facility allows omitting details at the beginning and thus permits early use of the specification language. In this way these AI ideas also yield some methodology for creating the specification.

A characteristic problem with formal specification languages is that they do not always take into account to a satisfactory extent that understanding of the problem grows only gradually during requirements understanding and specification, which altogether make these languages hard to use. On the other hand, experiences have shown that the development of more powerful and convenient specification languages, in itself, cannot be a solution to the specification problem. Their usage should also be supported by aids similar to those applied in current practice.

Practical specification methods lying within the scope of SE use such mixtures of formal and informal elements

in the specification that an optimal trade off between natural language ambiguity problems and formalization difficulties can be achieved. These methods usually facilitate top-down specification, viewing the system from multiple viewpoints, and other common analysis techniques which should be incorporated into any specification method.

ACKNOWLEDGEMENTS

We are indebted to I. Fekete, T. Asványi, T. Gregorics, S. Nagy, S. Sike and T. Vencel for their help in preparing this paper.

REFERENCES

[1] C. V. Ramamoorthy, A. Prakash, Wei-Tek Tsai and Y. Usuda, "Software Engineering: Problems and Perspectives", *IEEE Computer*, Oct. 1984, pp. 191-209.

[2] R. Balzer, T.E. Cheatham and C. Green, "Software Technology in the 1990's: Using a new Paradigm", *IEEE Computer*, Nov. 1983, pp. 29-45.

[3] R. Balzer, "A 15 Year Perspective on Automatic Programming", *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 11, Nov. 1985, pp. 1257-1268.

[4] S. Westfold, "Very-high-level programming of knowledge representation schemes", *AAAI-84*, Univ. of Texas, Austin 1984, pp. 344-349.

[5] E. Horowitz, A. Kemper and B. Narasimhan, "A Survey of Application Generators", *IEEE Software*, Jan. 1985, pp. 40-54.

[6] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 3, May 1982, pp. 250-269.

[7] P. Zave, "The Operational versus the Conventional Approach to Software Development", *Commun. ACM*, vol. 27, no. 2, Feb. 1984, pp. 104-118.

[8] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment", *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, Feb. 1986, pp. 312-325.

[9] Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems", *IEEE Softw.*, Sept. 1988, pp. 25-36.

[10] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Softw.*, March 1988, pp. 66-72.

[11] B.W. Boehm, "Software Engineering", *IEEE Computer*, vol. C-25, no. 12, 1976.

[12] A. Barr and E.A. Feigenbaum (Eds.), *The Handbook of Artificial Intelligence*, Vol. II, Chap. X, Automatic Programming, William Kaufmann Inc., Los Altos, CA, 1982.

[13] Z. Manna and R. Waldinger, *Synthesis: Dreams ==> Programs*, Rep. no. STAN-CS-77-630, Comp. Sci. Dept., Stanford University, 1977.

[14] D.R. Smith, G.B. Kotik and S.J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 11, 1985.

[15] S. Fickas and P. Nagarajan, "Critiquing Software Specifications", *IEEE Software*, Nov. 1988, pp. 37-47.

[16] D.R. Barstow, "Domain-Specific Automatic Programming", *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 11, 1985.

[17] J.R. Abrial, "Data Semantics", in J.W. Klimbie and K.L. Koffeman (Eds.), *Data Management Systems*, North-Holland, Amsterdam, 1974.

[18] J.R. Abrial, S.A. Schuman and B. Meyer, *Specification Language Z*. Massachusetts Computer Associates Inc., Boston 1979.

[19] D. Ince, "Z and System Specification", *Inf. and Softw. Tech.*, vol. 30, no. 3, April 1988, pp. 138-145.

[20] D. Bjorner and C.B. Jones, *Formal Specification and Software Development*., Prentice-Hall 1982.

[21] A. Andrews, "Specification aspects of VDM", *Inf. and Softw. Tech.*, vol. 30, no. 3, April 1988, pp. 164-176.

[22] R.M. Burstall, D.B. MacQueen and D.T. Sannella, "HOPE: an Experimental Applicative Language", in *Proc. 1980 LISP Conference*, Stanford 1980, pp. 136-143.

[23] R. Harper, R. Milner and M. Tofte, *The Definition of Standard ML*, Rep. no. ECS-LFCS-88-62, Univ. of Edinburgh, 1988.

[24] K. Futatsugi, J.A. Goguen, et al. "Principles of OBJ2", in *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans 1985, pp. 52-66.

[25] R.M. Burstall and J.A. Goguen, "Putting Theories together to Make Specifications" in *Proc. 5th Intl. Joint Conf. on Artificial Intelligence* (1977), pp. 1045-1058.

[26] D.T. Sannella, "A Set-Theoric Semantics for CLEAR", *Acta Inf.* 21, 1984, pp. 443-472.

[27] M. Wirsing, "Structured Algebraic Specifications: a Kernel Language", *Theoretical Computer Science* 42, 1986, pp. 124-249.

[28] D.T. Sannella, and A. Tarlecki, "Program Specification and development in Standard ML" in *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans 1985, pp. 67-77.

[29] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1*, EATCS Monographs on Computer Science, Springer 1985.

[30] P.P. Chen (Ed.), *Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, Amsterdam 1980.

[31] M.L. Brodie, "On the Development of Data Models" in M.L. Brodie, J. Mylopoulos and J.W. Schmidt (Eds.) *On Conceptual Modelling*, Springler-Verlag 1984, pp. 19-47.

[32] J. Mylopoulos, P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Interactive Database-Intensive Systems", *ACM Trans. Database Systems*, vol. 5, no. 2, June 1980, pp. 185-207.

[33] A. Albano and R. Orsini, "A Software Engineering Approach to Database Design: the Galileo Project" in A. Albano, V. de Antonellis and A. di Leva (Eds.), *Computer-Aided Database Design: The DATAID Project*, North-Holland, Amsterdam 1985, pp. 53-76.

[34] D.W. Shipman, "The Functional Data MOdel and the Data Language DAPLEX", *ACM Trans. Database Systems*, vol. 6, no. 1, March 1981, pp. 140-173.

[35] M. Hammer and B. Berkowitz, "Dial: a Programming Language for Data-Intensive Applications" in *Proc. ACM SIGMOD Conf.*, May 1980, pp. 75-92.

[36] A. Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level", *IEEE Software*, Jan. 1985, pp. 63-72.

[37] A. Borgida, S. Greenspan and J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications", *IEEE Computer*, April 1985, pp. 82-91.

[38] S. Greenspan, A. Borgida and J. Mylopoulos, "A Requirements Modeling Language and its Logic" in M.L. Brodie and J. Mylopoulos (Eds.) *On Knowledge Base Management Systems*, Springler-Verlag 1986, pp. 471-502.

[39] R.L. Glass, *Real-Time Software*, Prentice-Hall 1983.

[40] M. Goedicke, "Development of Realtime Systems: Specifying Functional and Parallel Behaviour Formally" *Proc. 14th IFAC/IFIP Workshop on Real-Time Programming*, Lake Balaton, Hungary 1986, pp. 67-79.

[41] B. Meyer, "On Formalism in Specifications", *IEEE Software*, Jan. 1985, pp. 6-26.

[42] J. Mylopoulos and H.J. Levesque, "An Overview of Knowledge Representation" in M.L. Brodie, J. Mylopoulos and J.W. Schmidt (Eds.) *On Conceptual Modelling*, Springler-Verlag 1984, pp. 3-17.

A Priori Information Support in the Classification of Satellite Images

Csornai, G., Nádor, G., dr. Dalia, O.
FÖMI Remote Sensing Centre
1149. Budapest, Bosnyák tér 5.

The agriculture plays key role in the Hungarian economy. Thirty percent of the total land is covered by different crops. That is why reliable, timely information on the major crops is of vital importance. The correct area estimation of the different crops, their stage and development assessment plus the reliable yield forecast are some of the main tasks of a national crop information system (CIS). This system should work real time and efficiently in terms of cost-benefit. The state of the art remote sensing - that is application of satellite images- provides a possible adequate tool to serve as a basis of the Hungarian CIS.

The satellites provide multispectral digital images of the Earth surface in a regular coverage pattern and with 5-16 days revisit interval. The multispectral image records the electromagnetic energy. This energy is quantized in different wavelength bands. That is the digital image is a matrix with vector elements (pixels). The surface element corresponding to one position in the image matrix is that is reflected by the objects (vegetation, soil, water bodies, etc.) on the Earth's surface 0.1-0.4 hectare. An image covers a 5000-36000 km² area on the ground. The spatial and temporal sampling capability of these images is superior to any of the existing field data collection methods.

The basic problem of those tasks outlined above is to derive a correct crop map plus area estimation figures using satellite

data. This can be done using image classification methods which comprise image processing and applied pattern recognition methods.

1. Experiences with per point and context dependent image classification methods

Pattern recognition techniques proved to be useful in analyzing multispectral satellite images. Different clustering (ISODATA type and histogram based) [Farkasfalvy, 1986] and classification (maximum likelihood and Bayesian) [Csornai et al, 1983a] methods has been applied. In the beginning the **context independent** techniques were used. These cluster or classify a pixel independent from its neighbours. On the Hungarian Great Plain we got fairly good results when developing and improved versions of these [Csornai et al, 1983a, 1983b]. The classification accuracies varied from 75-98% depending on the class. (The classes generally comprise more subclasses of wheat, corn, sugar-beet, sunflower, bare soil, etc.). Rigorous error estimation projects were done [Csornai et al, 1984] to analyze the impact of noise (field anomalies) and the tendency in misclassification. These accuracies were achieved on smaller (some farms) and bigger (1000-2000 km^2) areas, too. Lately we got a good crop map and area estimation figures 2-5% close to those of the Hungarian Statistical Office for a whole 0.6 million hectare county, Hajdú-Bihar [Csornai et al, 1987].

The **context dependent** methods make use of the strong correlation of vectors in the neighbouring pixels. Some methods build new features of the computed measures of the textural correlations [Haralick et al, 1973] and use them together with the spectral variables [Swain, 1976]. Others try to segment the image first, delineating homogenous blocks using some criteria [Robertson et al, 1973]. The ECHO, extraction and

classification homogeneous objects and its supervised (SECHO) version [Kettig, Landgrebe, 1976] are based on the facts, that objects (agricultural fields, ponds, etc.) are several times larger than the pixel size plus these pixels have similar intensity values. The original algorithm was improved [Fekete, Farkasfalvy, 1989] and installed onto a Microvax compatible processor.

Each of the context dependent algorithms has a basic assumption (model) on the relationship of the neighbouring pixels. The models generally are not specific to directions or particular locations, coordinates on the ground/image, therefore can work moderately well with different images. From these two important groups of context dependent analysis and classification algorithms the first (textural classification) has a weaker, less specific assumption than the second (homogeneous segments oriented).

Further increase in correct classification can be achieved if more a priori information is introduced into the classification scheme. There are many possible ways for that. A common example is when we use a priori class probabilities in the maximum likelihood decision. In case of Bayesian decision the a priori knowledge does not refer either to the relative frequency or the distribution of the classes, but to the loss arising from the misclassifications. In both cases these a priori data influence the result of the classification. The next reasonable step is to make use of the location and shape a priori information of the investigated objects.

2. The use of the field boundaries in image classification

Within the Hungarian circumstances the annual change rate of the agricultural fields boundary is estimated to be within the 2-5% interval. Thus if once the coordinates of these boundaries

are in a comprehensive data base (DFBD - digital field boundaries data), the image classification can be enhanced in several ways. Two of these have been studied at the FÖMI Remote Sensing Centre in details.

Prior to the image classification the DFBD is compared with the actual image to reveal the possible boundary changes. This is done in three steps. First the geometrical match between the DFBD and the image is accomplished using resampling techniques of the image [Dalia, 1983] or the polygon system of the field boundaries is mapped into the image geometrically. Some derivation operator (e.g. Roberts, Soebel, Kirsch) is then applied to extract the locations of the bigger intensity changes - the boundaries - from the image [Nádor, 1987] next. In the last step the DFBD and the derivation image is compared to locate the changes. (This is done now, visually though in a computer aided manner.) The DFBD is then updated. An example for DFBD can be seen in Fig.1.

The first method, say method A, starts with the per-point classification (e.g. maximum likelihood) and digitally superimposes the DFBD to the classified image. The procedure investigates the classified image within every field and based on the class relative frequencies the particular field is assigned to that of the biggest frequency (the most probable class). There are certainly some heterogeneity checks for the field included in addition. Method A reclassifies the fairly homogeneous fields while the rest are left with the per-point classification result.

Method B [Farkasfalvy, 1987] uses the DFBD at the very beginning of the procedure to compute second order statistics from the homogeneous fields. These fields are then classified using clustering and maximum likelihood method. Here the unit is the field instead of the pixel.

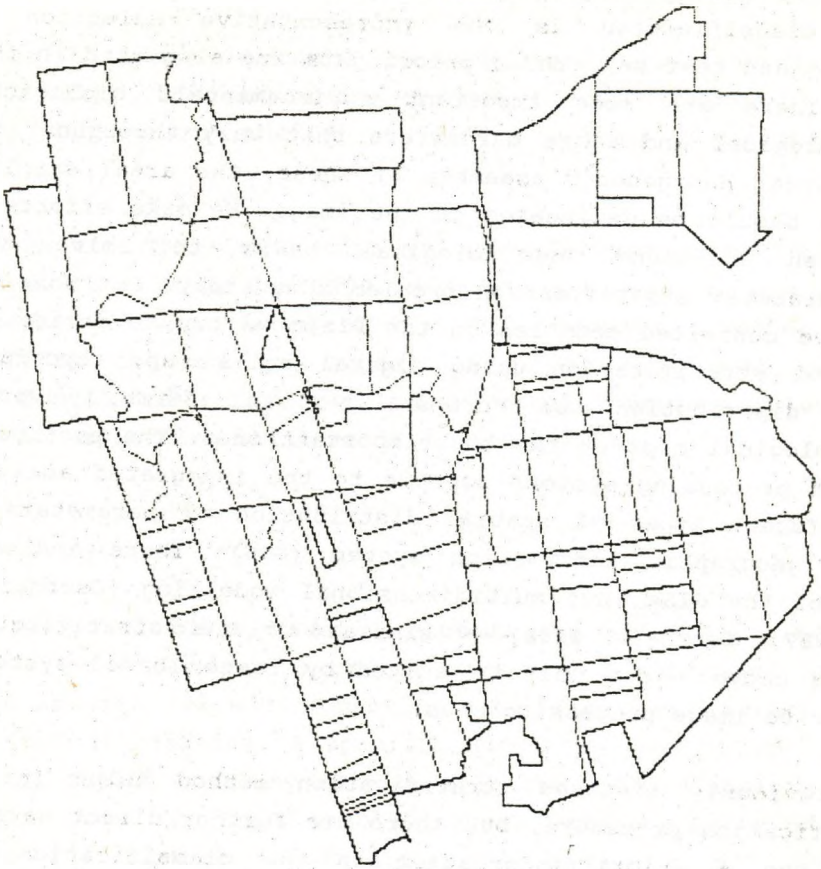


Figure 1: Digitized field boundaries of a cooperative, Hajdúböszörmény

These methods have performed quite well and increased the average percent correct classification by about 10, compared to the per-point method.

3. Further research on a priori information support

As one of the major problems of this supervised learning in image classification is the representative selection of training and test set, the a priori data are also used in this step. There are some important environmental, biological, climatological and other parameters that vary throughout the study area. Homogeneous assembly of these, the areal sampling stratum should be delineated on the image. We make efforts to establish a sound methodological basis to solve this multiparameter stratification problem adequately. For some two or three connected counties on the Plain we try to verify our model of stratification using digital soils map, topography maps, distribution of crops in the farms, average meteorological maps as the most important ones. The particular type of processing systems capable to the integrated analysis of different types of spatial distribution of parameters is called geographic information system (GIS). There are many types of the GISs, but multidimensional modelling [Csornai et al, 1987; Csornai, 1988] - similar to the stratification problem above - can only be solved by raster based systems, similar to image processing ones.

The previously sketched stratification method helps in the classification procedure, but there are further direct ways in using the a priori information in the classification. An example for this is based to a classification (per-point or other) plus a sequence of the check of rules derived from the agricultural practice and restrictions within a particular farm (e.g. the rule for corn → sunflower prohibited transition).

In the framework of a national crop information system that is organized in a GIS way the image classification procedure from year to year or within a season can rely on the series of results of the previous image classifications plus other (e.g. ground collected) ancillary data. If the classification model

(CM) is comprehensive enough, it can conceptually be supposed that the image classification errors tend to decrease in time that is reliable CIS could evolve.

The reliability holds until the background CM's assumptions are still valid. When there are changes in the components of the rules that control the CM, at least a warning is expected from the CIS.

Through the GIS system with strong modelling capabilities is planned to work by mid 1990, some valuable options are at our disposal already [Nádor, 1988]. The adequate computer aided stratification method is hoped to be accomplished soon, but the introduction of some agricultural practice derived rules into the classification still needs a lot of work.

4. Summary

The per-pixel classification methods can effectively be used for satellite images to create crop maps in Hungary. However there are many ways of improving classification accuracy using a priori information. A specific Hungarian possibility to use digital fields boundary data base in the image classification. Both methods that used DFBD increased accuracy by 10 percent. There are further possibilities to make use of a priori information with the use of GIS. Either the computer aided stratification or a learning CIS are good examples for the potential of strong reliance onto the a priori information.

Acknowledgement

The results reported in this paper were achieved in a R+D project having been supported by the State Board for Technical Development, the Ministry of Agriculture and Food and the Intercosmos Council of HAS.

References:

Robertson, T.V., Fu, K.S., Swain, P.H.: Multispectral image partitioning, Ph. D. Thesis #25970, School of Electrical Engineering, Purdue University, West Lafayette, IN, August, 1973.

Kettig, R.L., Landgrebe, D.A.: Classification of multispectral image data by extraction and classification of homogeneous objects, IEEE Trans. Geoscience Electronics, Vol. GE-14, No.1, January, 1976.

Haralick, R.M., Shanmugam, K., Dinstein, I.: Textural features for image classification, IEEE Trans. Systems, Man and Cybernetics, Vol. SMC-3, November, 1973.

Swain, P.H.: Land use classification and mapping by machine-assisted analysis of Landsat multispectral scanner data, LARS Information Note 111276, Purdue University, West Lafayette, Indiana, September, 1976.

Csornai, G., Dalia, O., Gothár, Á., Vámosi, J.: Classification Method and Automated Result Testing Techniques for Differentiating Crop Types, Proc. Machine Processing of Remotely Sensed Data, West Lafayette, USA, 1983a

Csornai, G., Vámosi, J., Dalia, O., Gothár, Á.: Vegetation Status Assessment and Monitoring in Agricultural Areas by Remote Sensing, XXXIVth IAF Congress, Budapest, 1983b

Csornai, G., Dalia, O., Gothár, Á.: Application results of efficient classification methods for multispectral images (in Hungarian), V. "Földfotó" Seminar, Budapest, 1984.

Csornai, G., dr. Dalia, O., Farkasfalvy, J., Nádor, G., dr. Vámosi, J., Zabó, P.: Development of assessment methods for the majcr crops on large area (in Hungarian), Research Report, FÖMI, Budapest, 1987.

Csornai, G., dr. Dalia, O., Farkasfalvy, J., Nádor, G., dr. Vámosi, J., Zabó, P.: Development of assessment methods for the major crops on large area (in Hungarian), Research Report, FÖMI, Budapest, 1988.

Csornai, G.: Satellite data based geographic information system for agricultural applications (in Hungarian), Geodézia és Kartográfia, Vol.40, No.4, 1988.

Dalia, O.: Geometrical transformation of digital images (in Hungarian), Geodézia és Kartográfia 1983/1.

Nádor, G.: Interband per-point transformations in spatial data analysis (in Hungarian), in: Csornai et al, 1988.

Nádor, G.: Program for edge detection (User's guide, in Hungarian), FÖMI, Budapest, 1987.

Farkasfalvy, J.: Clustering methods for remotely sensed data, 2nd Conference of Program Designers, ELTE, Budapest, 1986.

Farkasfalvy, J.: Comparison of the per-point and per-field classification methods in the processing of remotely sensed data, 3rd Conference of Program Designers, ELTE, Budapest, 1987.

Fekete, I., Farkasfalvy, J.: Automatic segmentation of multispectral digital images, Conference of Artifical Intelligence, ELTE, Visegrád, Jan.1989.

COMPARISON OF DIFFERENT CLASSIFICATION METHODS USING LANDSAT TM DATA

J. Farkasfalvy
FÖMI Remote Sensing Centre
1149. Budapest, Bosnyák tér 5.

Different methods, which we have developed for classification, are compared on a common subimage of a Landsat TM scene. The result of this case study for a given image in a given date does not qualify the methods sufficiently. It is necessary to examine them for different cases (area, date), and dependence on the data and parameters has to be recognized. These experiments have been started [5].

Image to be used was: Landsat TM, date: July 8, 1987. Four TM bands were used in the process. The selected area is on Hungarian Great Plane, the image matrix[1] consists of 512 by 512 pixel. The vegetation on this area had been known, so we could complete the digital reference map that was used in the classification procedure.

Two traditional per-point procedures were used with different clustering methods: ISODATA [2, 3], and histogram-clustering [3]. There are two methods based on field boundaries, we call them method A and method B for the sake of simplicity.

The **method A** [1, 6] investigates the result of per-point classification: computes the class relative frequencies for every field. If a field consists of mostly one class' pixels, all the pixels will be ranged to this class. This method is a smoothing of the classified image.

---

[1]A pixel is a vector, containing reflected radiances in different electromagnetic wavelength regions, from a given area of the Earth's surface.

The **method B** [4, 5] uses the field boundaries at the very beginning of the classification. First the second order statistics of fields are computed, then the fields having not too large variance are clustered and classified by per-field methods, using their stat. data. The pixels not belonging to any homogeneous fields are processed by per-point methods.

The aim of developing of the methods based on reference data is increasing the classification accuracy, by filtering some local errors of few pixels (in-field inhomogeneities).

1. Components of the 1st per-point classification plus method A

- **Clustering by ISODATA method:** Clustering was done for a subsample. The reason was that some categories could not be separated after clustering the whole image, because of the unsuitable area ratio of the categories. Number of clusters was 30 at the beginning, and 25 at the end of the process.

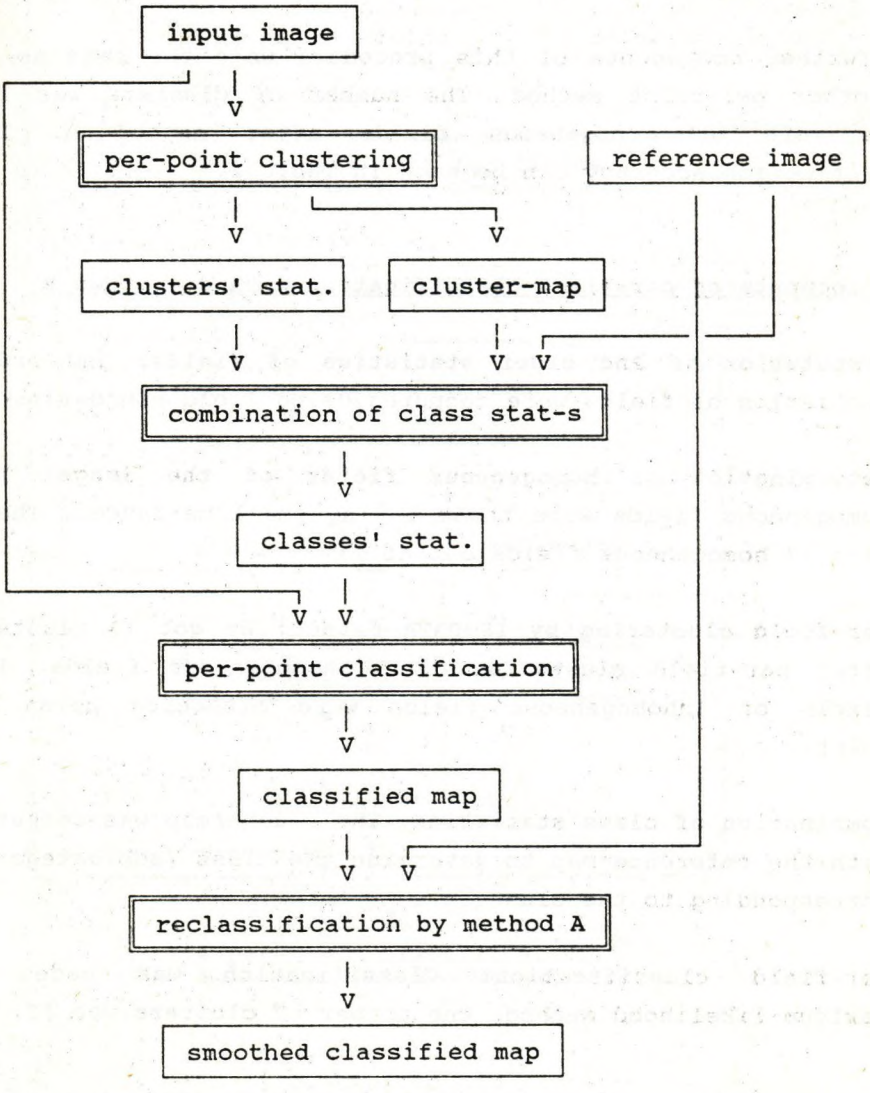
- **Combination of class statistics:** The clustermap was compared with the reference map to determine the cluster components of the categories. The classes' statistics were computed by mixing the clusters' density functions.

- **Maximum-likelihood classification:** Classification was made by maximum-likelihood method, the number of classes was 13.

- **Computation of classification accuracy:** The classification results was compared to the reference map, and we got the classification accuracy (contingency) matrix.

- **Reclassification by method A:** The result of the per-point classification was smoothed. The parameters were the following: if the relative frequency of a class was greater than 60% in a field, and there were not any class having

relative frequency greater than 20%, all the pixels of this field would be ranged to the dominant class. There were 53 homogeneous fields after method A. The classification accuracy increased (see Table 2.).



Components of per-point classification + method A

2. Components of the 2nd per-point classification plus method A

- **Histogram-clustering:** Only 3 bands could be processed in histogram clustering. The best combination of bands was chosen. Clustering was done by partitioning this 3-dimensional histogram. Number of clusters was 34.

The further components of this procedure were the same as in the other per-point method. The number of clusters was 13. There were 68 homogeneous fields after method A. The classification accuracy can be seen in Table 2.

3. Components of per-field classification method (method B)

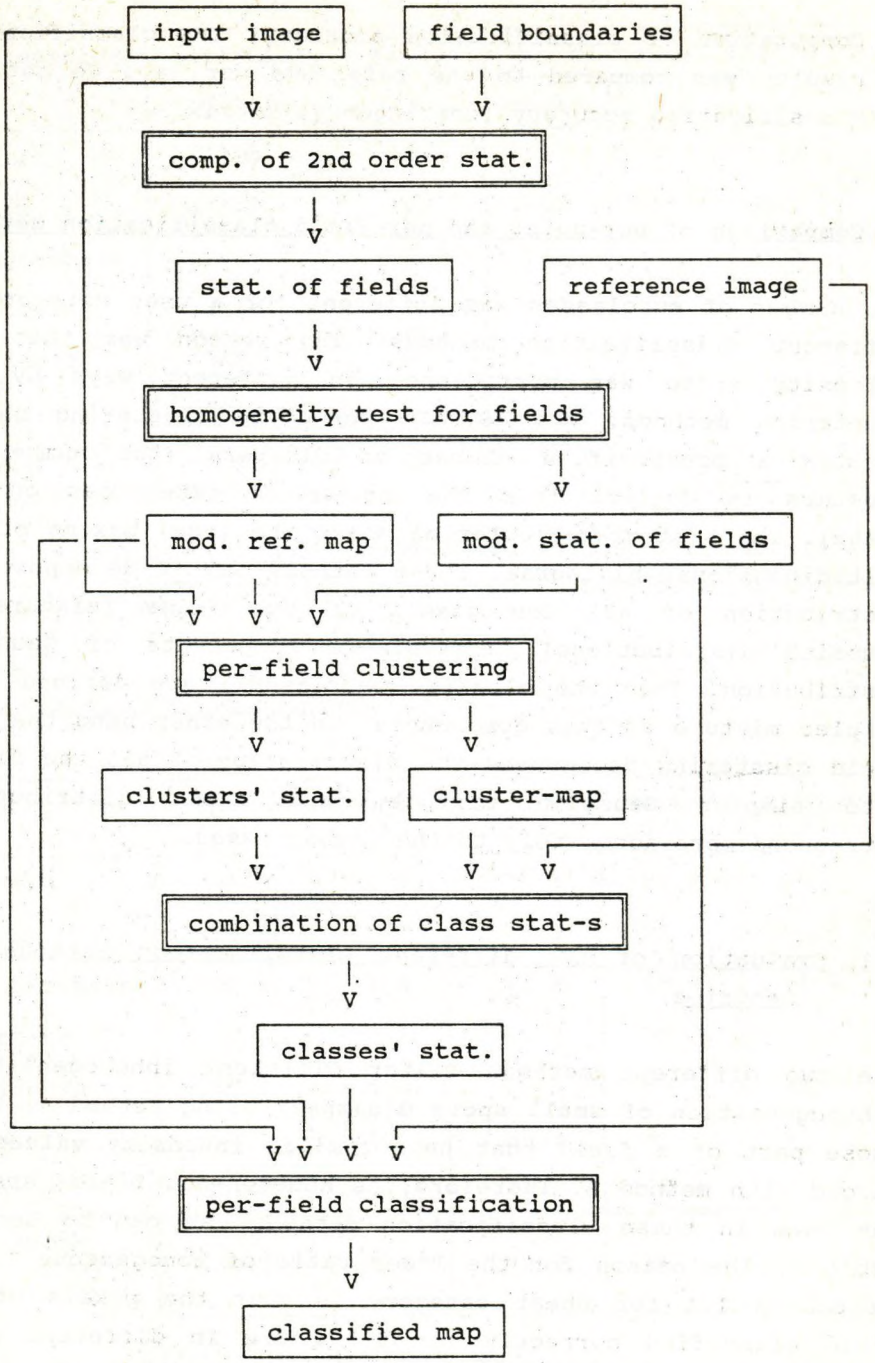
- **Computation of 2nd order statistics of fields:** 2nd order statistics of fields were computed using field boundaries.

- **Determination of homogeneous fields of the image:** The homogeneous fields were those having small variances. There were 77 homogeneous fields out of 117.

- **Per-field clustering by ISODATA method:** We got 13 clusters after per-field clustering of 77 homogeneous fields. The pixels of inhomogeneous fields were clustered point by point.

- **Combination of class statistics:** The clustermap was compared with the reference map to determine the class (sub-category) corresponding to the clusters.

- **Per-field classification:** Classification was made by maximum-likelihood method, the number of clusters was 13.



Components of method B

- **Computation of classification accuracy:** The classification results was compared to the reference map, and we got the classification accuracy (contingency) matrix.

4. Comparison of per-point and per-field classification methods

The number of subclasses was different for a user category in different classification methods. The reason was that the intensity space was partitioned in different ways by the clustering methods. The ISODATA per-point clustering method locates a prespecified number of clusters (the number of clusters is derived from the number of categories on the image). The histogram-clustering seeks the local maxima of the multidimensional histogram. These methods try to decompose the distribution of all the pixels of the image (mixture of Gaussian distributions) into simple components of Gaussian distribution. Then the classes (subclasses) are defined as a simpler mixture of this components. On the other hand the per-field clustering decomposes the distribution of all the pixels into simpler Gaussian mixture, and these distributions correspond more adequately to the (sub)classes.

4.1. Evaluation of the different classification methods per categories

The two different methods filter different inhomogeneities: inhomogeneities of **small spots** disappear using method A, while those part of a field that have **similar intensity values** are merged with method B. Therefore the homogeneous fields are not the same in these classification methods, as can be seen in Table 1. The reason for the lower ratio of homogeneous fields in column 1.A for wheat category is that the pixels of the field classified correctly in 80-90% are in different wheat subclasses.

Categories	# pixels	# fields	No. of homog. fields		
			1.A	2.A	B
Wheat	41150	38	11	30	36
Corn	37442	40	19	18	25
Sugar-beet	12485	13	9	5	3
Potato	4186	7	5	6	2
Alfalfa	3687	6	5	4	5
Soil	2026	4	3	2	2
Water	3568	2	2	2	0

Table 1: Number of homogeneous (classified with accuracy of 100%) fields in the different classification methods

4.2. Classification accuracy

Categories	1/point class	Method 1/A	2/point class	Method 2/A	Method B
Wheat	87 %	91 %	78 %	89 %	98 %
Corn	73 %	78 %	79 %	82 %	90 %
Sugar-beet	65 %	83 %	61 %	67 %	81 %
Potato	67 %	83 %	76 %	87 %	77 %
Alfalfa	65 %	85 %	77 %	81 %	96 %
Soil	87 %	91 %	83 %	88 %	45 %
Water	71 %	100 %	71 %	100 %	76 %
Average accuracy in percent	80.1	86.8	78.9	85.6	91.6

Table 2: The classification accuracies of the different methods

It is obvious from the table that the accuracy for method A exceed the accuracy of simple per-point methods, for some categories significantly. The average accuracy is the best for method B, and it is low only for soil category because of the inexact reference data.

The total processing time was the shortest in method B, and it would have been even much shorter, if more pixels had belonged to the homogeneous fields, because the great part of the pixels of image was processed per point in this case too.

5. Summary

Different image classification methods were compared. Their major steps were summarized. The methods that used digital field boundaries increased the classification accuracy. In this case study the method B seemed to be the most efficient and accurate.

Acknowledgement

The results reported in this paper were achieved in a R+D project having been supported by the State Board for Technical Development, the Ministry of Agriculture and Food and the Intercosmos Council of Hungarian Academy of Sciences.

References:

[1] **G. Csornai, dr. O. Dalia, J. Farkasfalvy, G. Nádor, dr.J. Vámosi, P. Zabó:** Development of assessment methods for the major crops on large area (in Hungarian), Research Report, FÖMI, Budapest, 1987.

[2] **O. Dalia:** Evaluation of classification methods for remotely sensed multispectral images (in Hungarian), Doctorial dissertation, Technical University, Budapest, 1986.

[3] **J. Farkasfalvy:** Clustering methods for remotely sensed data. 2nd Conference of Program Designers, ELTE, Budapest, 1986.

[4] **J. Farkasfalvy:** Comparison of the per-point and per-field classification methods in the processing of remotely sensed data. 3rd Conference of Program Designers, ELTE, Budapest, 1987.

[5] **J. Farkasfalvy:** Some problems of parameter selection in a complex procedure of per-field classification. 4th Conference of Program Designers, ELTE, Budapest, 1988.

[6] **G. Nádor:** Utilization of ancillary data in the classification of agricultural fields (method A) (in Hungarian), FÖMI, Budapest, 1987.

AUTOMATIC SEGMENTATION OF MULTISPECTRAL DIGITAL IMAGES

I. Fekete[1], J. Farkasfalvy[2]
[1] ELTE Dept. of Computer Sciences
[2] FÖMI Remote Sensing Centre

INTRODUCTION

The remotely sensed multispectral images deliver information from a given area of the Earth's surface. The data can be considered as an image matrix, containing vector elements. The neighbouring elements of the matrix correspond to neighbouring spots on the surface, so the close picture elements (pixels) are similar. This similarity is utilized, when segments are created from neighbouring pixels, and these are processed in the classification algorithms as a unit instead of the pixels. The processing of the arrays of statistically similar pixels provides better classification, because some local errors of few pixels are filtered. The number of segments is much less than the number of pixels, so the processing time is reduced, too.

THE ALGORITHM FOR SEGMENTATION

The algorithm can be divided into two levels.
First the image is partitioned into rectangles of k*l pixels. These rectangles are called cells, and they are the units of the segmentation. The cells are small, for example k=l=2. The cells are tested if they satisfy the criterion of homogeneity or not.
In the next level the neighbouring cells are merged, if they are homogeneous and statistically similar by some appropriate criterion. The group of the connected cells is called a segment.

Homogeneity test for the cells

A cell is homogeneous, if the ratio of the square root of the sample variance to the sample mean is lower than some threshold in any channel.

Let

$$c_j = \frac{\sigma_j}{x_j} = \frac{1}{x_j} \left(\frac{1}{n-1} \sum_{i=1}^{n} x_{j,i}^2 - \frac{n}{n-1} x_j^2 \right)^{\frac{1}{2}}, \quad j=1,\ldots,r$$

where x_j is the mean value of the pixels of a cell in the jth channel, $x_{j,i}$ is the intensity value of a pixel in this channel.

A cell is homogeneous, if $c_j \leq c_H$, in every channel $j=1,\ldots,r$, where c_H is the prespecified threshold value.

Annexation of a homogeneous cell to a neighbouring segment

The goal is merging the statistically similar cells into segments. The procedure starts from the upper left corner of the image, goes forward rowwise, and takes only homogeneous cells into consideration. The annexation criteria is a statistical hypothesis testing.

Let $X = (X_1,\ldots,X_n)$ represent the pixels in a group of cells have been merged by successive annexation. Let $Y = (Y_1,\ldots,Y_n)$ represent the pixels in a homogeneous cell. X and Y is assumed to have multivariate normal density function. The cell Y and group of cells will be merged, if the corresponding density functions are the same, or are close to each other.

The following quantities are computed for every channel:

$L_1 = (A/B)^{N/2}$

$$L_2 = \left\{ \frac{(A_x/n)^{n-1} * (A_y/m)^{m-1}}{(A/N)^N} \right\}^{\frac{1}{2}}$$

where $N = n+m-2$,

furthermore

$$x = \frac{1}{n} \sum_{i=1}^{n} x_i \qquad y = \frac{1}{m} \sum_{i=1}^{m} y_i$$

$$A_x = \sum_{i=1}^{n} (x_i-x)^2 = \sum_{i=1}^{n} x_i^2 - n*x^2 \qquad A_y = \sum_{i=1}^{m} (y_i-y)^2 = \sum_{i=1}^{m} y_i^2 - m*y^2$$

$$A = A_x+A_y$$

$$M = (nx+my)/(n+m)$$

$$B_x = \sum_{i=1}^{n} (x_i-M)^2 = \sum_{i=1}^{n} x_i^2 - n*M^2 = A_x + n(x-M)^2$$

$$B_y = \sum_{i=1}^{m} (y_i-M)^2 = \sum_{i=1}^{m} y_i^2 - m*M^2 = A_y + m(y-M)^2$$

$$B = B_x+B_y$$

The Y cell will be connected to the group of cells X, if $L_{j,1} \geq c_1$ and $L_{j,2} \leq c_2$, in all channels $j=1,\ldots,r$, where c_1 and c_2 are threshold values.

Evaluation of L_1 and L_2 is independent. L_1 tests the hypothesis of equal mean vectors (first order statistics), and L_2 tests the hypothesis of equal covariance matrices (second order statistics). When Y and X are merged, the mixture of their density functions is computed to be the new density function of the merged samples of the group of cells X.

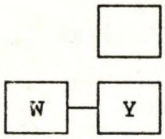


Figure 1/a.

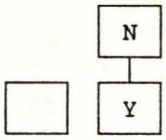


Figure 1/b.

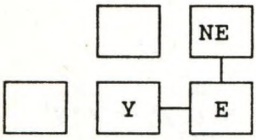


Figure 1/c.

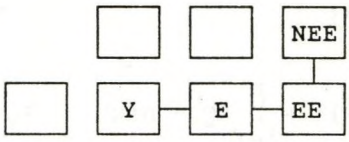


Figure 1/d.

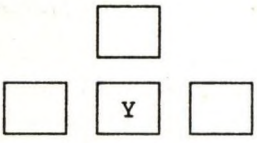


Figure 1/e.

The procedure tries to merge the cell to its west or north-adjacent cell (Figure 1/a, 1/b). These cells, if they are homogeneous, have already been assigned to a partition because of the processing sequence. If the cell can be merged both of north and west neighbouring partition, and they are not the same, the algorithm choose one of them on the basis of a distance function. The cell is merged to that partition, mean vector of which is closer to the cell's mean vector in Euclidean distance.

If the cell cannot be connected to any of these adjacent cells, the algorithm investigates the east neighbours to find the way to a partition having been started (Figure 1/c, 1/d). The annexation criteria is examined for the east or second east neighbouring cell and its north adjacent cell. If the connection can be done, the cell Y is tested to be merged to

this partition. This is an important improvement of the algorithm, because in this way the segmentation is symmetric, the segments can expand to the right, too. If the cell cannot be connected to any adjacent cells, it starts a new field itself (Figure 1/e).

For the memory management the segments are examined after processing a cell-row. If a segment cannot be continued, its data (first and second order statistics) is written into stat. file and can be deleted. Those segments come to the end in a row, which exists in previously processed row and disappear in the examined row.

Parameter selection

The segmentation algorithm has three essential parameters mentioned above:

- upper bound for cells' homogeneity: c_H,
- lower bound for annexation of a cell by one of its neighbouring segment: c_1,
- upper bound for annexation of a cell by a segment: c_2.

When increasing the value of c_H, the number of homogeneous cells increases. If the value of c_1 decreases and/or the value of c_2 increases, there will be more homogeneous cells belonging to a segment, therefore fewer segments of larger size will be created.
Good parameter selection can be done by an iterative test.

SUMMARY

The unsupervised algorithm for segmentation makes use of the strong correlation of vectors in the neighbouring pixels. Some modifications to the basic algorithm of segmentation was done.

These improvements are: choice between north and west-adjacent partitions using distance functions, investigation east neighbouring cells and memory management.

REFERENCES:

R.L. Kettig, D.A. Landgrebe: Classification of multispectral image data by extraction and classification of homogeneous objects. IEEE Transactions on Geoscience Electronics, Vol. GE-14, No. 1. January 1976.

I. Fekete: Segmentation, clustering and classification by segments, in: Programs for classification of multispectral remotely sensed digital images. Methods and program schemes, Guide for development (in Hungarian). ELTE, Budapest, 1983.

J. Farkasfalvy: Comparison of the per-point and per-field classification methods in the processing of remotely sensed data. 3rd Conference of Program Designers, ELTE, Budapest, 1987.

J. Farkasfalvy: Some problems of parameter selection in a complex procedure of per-field classification. 4th Conference of Program Designers, ELTE, Budapest, 1988.

Declarative and Procedural Style of Logic Programming

Tibor Ásványi
(ELTE, Budapest, February, 1989.)

Abstract

This paper is about declarative and procedural use of Horn clauses.
My special problem is how to make a PROLOG program from a set of Horn clauses. Such a set is an exact specification of the problem. After all, it is not a method or an algorithm to solve the problem. It only describes the problem.

Particulary, I deal with path-finding. It is not trivial to make a bridge through the gap between declarative and procedural attention of path-finding problems, even if the search space is finite.

Introduction

We can make PROLOG programs in the following way :

1, Describe the problem using Horn clauses.

I recommend this implication form:
- Our (problem specific) assertions have the form of A <- .
- Our (general purpose) rules (procedures) have the form of B <- C1, C2, ... , Cn. (n >= 0)
- Our goal statement have the form of <- D1, D2, ... Dm. (m > 0)
(A, B, C1, ... , Cn, D1, ... , Dm are atomic formulaes.)

2, We consider it a specification.

Then make up a theorem-proving system (control component) to solve the problem using the clauses above.
(Logic program = Logic + Control)

3, If it is the same as the PROLOG control system, it means success: STOP!

4, If not, the control system can be approached to the PROLOG control system, simulate the previous solution by reformulating the problem.

5, Go to 3.

Path-finding

problems can be expressed as follows:

'Given an initial state A , a goal state Z and operators wich transform one state into another, the problem is to find a path from A to Z .' (.1.)

The water containers problem (.1.)

Given both a 7 and a 5 litre container, initially empty. Three kinds of actions are allowed:

1, A container can be filled.

2, A container can be emptied.

3, Liquid can be poured from one container into the other, until the first is empty or the second is full.

The goal is to find a sequence of actions wich leaves 4 litres of liquid in the 7 litre container.

If the water containers problem is considered as a path-finding problem, a simple Horn clause formulation can be used.

Interpret

State(u,v)

the state with u litres of liquid in the 7 and v litres of liquid in the 5 litre container.

Let the

z=x+y , z=x+y-u , x<=y and x>y

relations be given, as usually, on the integer type.

Kowalski's Horn clause formulation is trivial:
(Lower-case letters (maybe indexed) are variables, digits and other symbols are constants on the places of simple arguments.)

```
wc1   State(0,0) <- .                      (*Initial state*)
wc2   <- State(4,b) .                      (*Goal state*)

wc3   State(7,y) <- State(x,y) .           (*Filling *)
wc4   State(x,5) <- State(x,y) .           (* a container*)

wc5   State(0,y) <- State(x,y) .           (*Emptying *)
wc6   State(x,0) <- State(x,y) .           (* a container*)

wc7   State(0,y) <- State(u,v) ,           (*Pouring from *)
                    y=u+v , y<=5 .         (* one container into *)
wc8   State(x,0) <- State(u,v) ,           (* the other until *)
                    x=u+v , x<=7 .         (* the first is empty*)

wc9   State(7,y) <- State(u,v) ,           (*Pouring from *)
                    y=u+v-7 , y>0 .        (* one container into *)
wc10  State(x,5) <- State(u,v) ,           (* the other until *)
                    x=u+v-5 , x>0 .        (* the second is full*)
```

This set of clauses can be easily coded in PROLOG but its running makes failures:

1, It makes an infinite recursion at the wc4 clause.

2, Even if we could eliminate the infinite recursions, the running of the program would fail because of the top-down control system: For example it could not solve the subgoal: y=u+v . (Both u and v are undefined and it raises a program error.)

Moreover a natural solving process of the problem does not go from the goal to the initial state, but from the initial state to the goal (for instance) as follows:

Program development 1.

0, Let the initial state be at the place of the 'last generated assertion'.

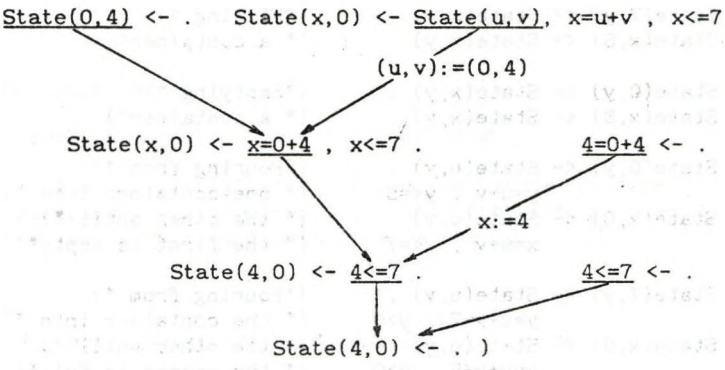
1, Take the 'last generated assertion'.
Try to match it with the goal statement.

2, If it can be done, the program is finished: STOP!

3, Otherwise:

4, Match it with the first rule wich has not been tried.
Then try to eliminate the other conditions
(if they are exist) .
In this way you may get the next assertion.

(For example:



5, If it is successful

A, and we have generated an assertion different from the previous ones, let it be the 'last generated assertion' and go to 1.

B, Otherwise reject the new assertion and take the previous one. Go to 4.

6, If the other conditions cannot be eliminated take the 'last generated assertion' again and go to 4.

7, If all the rules have been tried, reject the actual assertion and let the 'last generated assertion' be the previous one. Then go to 4.

8, If there is no previous assertion then there is no solution. STOP!

Notes 1.

1, The solution found here is provided by the assertions with the form of 'State(.,.) <- .' on the way of the refutation found here.

2, The algorithm 0-8 above is appropiate in all those cases when the state space is finite. (Here are at the most 2*8+2*6-4=24 states in the state space because one of the containers is allways empty or full.)

3, If one point of the state space can be described by n arguments, it can be representated using a predicate
State(x1,x2,...,xn) .

Program development 2.

In order to approach to the PROLOG level of the algorithm (logic program) above we should simulate it using a top-down (goal-controlled) resolution (refutation) system.

We will use the problem solving interpretation of Horn clauses. For example the rule (prodedure) ' A <- B , C . ' is interpreted as follows:
To solve A , solve B and then C .

Interpret St(x,y,a,b) the goal, that you have to get from State(x,y) to State(a,b) .

```
vc0     <- State(4,b) .
vc1     State(a,b) <- St(0,0,a,b) .
vc2     St(a,b,a,b) <- .

vc3     St(x,y,a,b) <- St(7,y,a,b).
vc4     St(x,y,a,b) <- St(x,5,a,b).

vc5     St(x,y,a,b) <- St(0,y,a,b).
vc6     St(x,y,a,b) <- St(x,0,a,b).

vc7     St(u,v,a,b) <- y=u+v , y<=5 , St(0,y,a,b).
vc8     St(u,v,a,b) <- x=u+v , x<=7 , St(x,0,a,b).

vc9     St(u,v,a,b) <- y=u+v-7 , y>0 , St(7,y,a,b).
vc10    St(u,v,a,b) <- x=u+v-5 , x>0 , St(x,5,a,b).
```

Where our interpreter differ from PROLOG one only in the fact that it makes a back-track from a goal statement of the form ' <- St(.,.,.,.) . ' if it is the same as one of its ancestors, as well as in the case of unsuccessful solution.

Notes 2.

1, We suppose a PORLOG interpreter wich is looking for only one solution.

2, Notice that if you could reformulate wc1-wc2 using vc0-vc2 , wc3-wc10 rules interpreted bottom-up can be reformulated step by step as vc3-vc10 rules interpreted top-down.

3, The solution is provided by the goal statements of the form <- St(.,.,.,.) . , on the way of the refutation.

Program development 3.

There are two problems to solve:

1, Not to generate a goal of the form <- St(.,.,.,.) . , wich has been generated (as an ancestor) .

2, Providing an explicite output.

We can solve the first problem if we change St into a five place predicate. (It will have the same name.) The first four places play the same role. In the fifth one we can collect the states achieved earlier (as ancestors) .

A state can be represented by a term s(.,.) .

Consider the program bellow, as if it were withouth the last place in each of the atomic formulaes St and State .

```
c0      <- State(4,b,1) .
c1      State(a,b,1) <- St(0,0,a,b, s(0,0).nil ,1) .
c2      St(a,b,a,b,1,1) <- .


c3      St(x,y,a,b,h,1) <- Not(Elem( s(7,y) ,h)) ,
                           St(7,y,a,b, s(7,y).h ,1) .
c4      St(x,y,a,b,h,1) <- Not(Elem( s(x,5) ,h)) ,
                           St(x,5,a,b, s(x,5).h ,1) .

c5      St(x,y,a,b,h,1) <- Not(Elem( s(0,y) ,h)) ,
                           St(0,y,a,b, s(0,y).h ,1) .
c6      St(x,y,a,b,h,1) <- Not(Elem( s(x,0) ,h)) ,
                           St(x,0,a,b, s(x,0).h ,1) .
```

```
c7      St(u,v,a,b,h,l) <- y=u+v , y<=5 ,
                           Not(Elem( s(0,y) ,h)) ,
                           St(0,y,a,b, s(0,y).h ,l) .
c8      St(u,v,a,b,h,l) <- x=u+v , x<=7 ,
                           Not(Elem( s(x,0) ,h)) ,
                           St(x,0,a,b, s(x,0).h ,l) .

c9      St(u,v,a,b,h,l) <- y=u+v-7 , y>0 ,
                           Not(Elem( s(7,y) ,h)) ,
                           St(7,y,a,b, s(7,y).h ,l) .
c10     St(u,v,a,b,h,l) <- x=u+v-5 , x>0 ,
                           Not(Elem( s(x,5) ,h)) ,
                           St(x,5,a,b, s(x,5).h ,l) .


c11     Elem(x,x.y) <- .
c12     Elem(x,y.z) <- Elem(x,z) .


c13     Not(x) <- x , ! , FAIL .          (* x is a   *)
c14     Not(x) <- .                       (* predicate! *)
```

A possible solution of the second prolem is the program above as a whole.

Applying c2, we get a possible form of the output in the last place of St . (It is the reserved list of the way from s(0,0) to s(4,b) .) So the last places of St and State are needed to gain the output in the goal statement.

Notes 3.

1, Notice, if we write the c3-c10 clauses in the order c7-c10 , c3-c6 , we can increase the effectiveness of the PROLOG code.

2, One can argue against the program, that it tries to solve the goals of the form ' <- St(.,.,.,.,.,.,.) . ' already recognised (in another way) as unsolvable goals.

But it is a basic problem, because any PROLOG backtrack forgets the information generated by trying to solve a subgoal, when it is recognised as unsolvable and is rejected. Each of the possible solutions need new means.

A, One of the possible solutions is generating negative lemmas (.1.) , (.5.) . 'Negative lemmas ... need to be generated when a subgoal is recognised as unsolvable. Negative lemmas can be used to recognise that the same subgoal is unsolvable when it arises in another context.' (.1.)

B, One other possibility is a PROLOG, but no logical means: The usage of global variables wich are not changed when a backtrack is raised. All the states achieved, have been put into a global list.
We check if there is an Elem relation between the actual state and the list.

Summary

It is trivial now, that the notion of logic programs is not the same as the specification of problem using a well defined logic formalism. But it is not equal with any PROLOG .

We can see: Logic program = Logic + Control .

The specification of the problem is the abstract logic component.

The abstract program is the specification
(or a logically equivalent formulation of the problem)
+ a matching control component.

The development of a concrete logic program means to transforme these two components,
until the appropiate form is achieved, it can be written in an available or specially designed programming language (PROLOG) .

References

(.1.) Kowwalski, R.A.: Logic for Problem Solving.
North Holland, 1979.
(Amsterdam, New-York, Oxford)

(.2.) Lloyd, J.W.: Foundations of Logic Programming.
Springer-Verlag, 1984.
(Berlin, Heildelberg, New-York)

(.3.) Loveland, D.V.: Autamated Theorem Proving:
A Logical Basis.
North Holland, 1978.
(Amsterdam, New-York, Oxford)

(.4.) Nilsson, D.W.: Principles of Artificial Intelligence.
Springer-Verlag, 1981.
(Berlin, Heildelberg, New-York)

(.5.) Earley,J.: An Efficient Context-free Parsing Algorithm.
CACM, pp.94-102., 1970.

(.6.) Clockskin, W.F., Mellish, C.S.: Programming in PROLOG.
Springer-Verlag, 1981.
(Berlin, Heildelberg, New-York)

ANOTHER INTRODUCE TO CONSISTENT ALGORITHMS

Tibor Gregorics

Eötvös Loránd University
Department of General Computer Science
Budapest

ABSTRACT This paper defines a class of the graph-search algorithms making the monotone restriction wider within the class of the algorithms A. It is proved that these algorithms, we call consistent algorithms (algorithms A^C), are admissible and preserve their original properties. Another result of this paper is a proof that the class of the algorithms A^C is a subclass of the algorithms A^*.

1. INTRODUCTION

A number of problems in the Artificial Intelligence (AI) area can be related to the general problem of finding a path through a space of problem states from the initial state to a goal state. In this *state-space representation* any problem can be treated as a directed graph. To get a solution to that problem means thus to find a path in the graph from the start node to a goal node.

Severel search techniques have been developed, which use heuristic information, i.e. special knowledge available from the problem domain in order to solve this search problem in an efficient way. Among the *heuristic graph-search algorithms* we have the class of the algorithms A [Nilsson, 82] encompassing the class of the algorithms A^* [Hart, 68].

In this paper we are going to show that the monotone restriction [Nilsson, 82] can be made wider within the class of the algorithms A, the class of these algorithms is admissible and it can be encased in the class of the algorithms A^*.

2. GRAPH NOTATION

A graph is defined as a set of nodes N, and a set of arcs A. Each arc is directed from a node to another one. We use the notation c(n,m) to denote the cost of an arc directed from the node n to the node m. We assume that these costs are all higher than some arbitrarily small positive number δ. Each node of the graph of our research has only a finite number of arcs. Let T be the set of the goal nodes of the graph. We call these directed δ-graphs *representation graphs* [Handbook, 82].

The cost of a path from one node to another is the sum of the costs of all the arcs connecting the nodes on that path. We want to find the path of minimal cost between s and any member of the set T in any problem. We call this path the *optimal path*. We will use

$g^*(n)$ = cost of optimal path from s to n, if there exists a path between s and n.

$h^*(n)$ = cost of minimal path from n to the closest member of T, if there exists a path between s and T, otherwise $h^*(n) = \infty$.

$f^*(n)$ = cost of minimal path from s to the closest member of T constrained to include node n, i.e. $f^*(n) = g^*(n) + h^*(n)$.

The *algorithm A* with minor formal modifications is the following.

```
Procedure GRAPHSEARCH
    G ← {s}; OPEN ← {s}; g(s) ← 0; p(s) ← nil
    n ← s
    while not ( goal(n) or empty(OPEN) ) loop
        OPEN ← OPEN \ {n}
        M ← Γ(n)
        while not empty(M) loop
            m ← member(M)
            if m ∉ G or else g(m) > g(n) + c(n,m) then
                g(m) ← g(n) + c(n,m); p(m) ← n
                OPEN ← OPEN ∪ {m}
            endif
        endloop
        G ← G Ũ Γ(n)
        n ← min_f(OPEN)
    endloop
end GRAPHSEARCH
```

where

- G contains the actual subgraph of the representation graph built explicitly by the algorithm we call search graph.
- OPEN is the set of nodes which have not been expanded yet.
- Γ is the successor operator which generates all of successors of any node.
- Ũ moves all the successors of any node together their arcs into the search graph G.
- M is the set of the successors of any node.
- p is a pointer directed from a node back to any of its parents.
- g(n) is the cost of the path from s to n in G.
- f is the *evaluation function* which can be calculated for any node n as f(n) = g(n) + h(n) where the *heuristic function* h greater than or equal to the zero function.

We known that the algorithms A always find a path from start node s to a goal node (of T) if there exists a path between s and T [Fekete, 88]. When the algorithm A uses an h function that is a lower bound on h^* ($h \le h^*$), we call it *algorithm* A^*. This algorithm is *admissible*, i.e. it always finds an optimal path from the start node to goal nodes whenever this path exists [Hart, 68].

3. ALGORITHM A^c

We call the algorithm A *algorithm* A^c when the heuristic function h satisfies the monotone restriction (i.e. for all nodes n and m, where m is a successor of n, $h(n) - h(m) \le c(n,m)$), and $h(t) = 0$ for each goal node t.

Now we show that the algorithm A^c is admissible, i.e. if there is a path between the start node s and the set T the algorithm terminates by finding an optimal path.

Lemma 1. For any node n and m on an optimal path from s to n, where m is a descendant of n

$$g^*(n) + h(n) \le g^*(m) + h(m).$$

Let the ordered sequence $P = (s=n_0, n_1, \ldots, n_k=n)$ be this optimal path. Using the monotone restriction, we have the

$$g^*(n_i) + h(n_i) \le g^*(n_i) + c(n_i, n_{i+1}) + h(n_{i+1}) \qquad (i = 1,\ldots,k-1)$$

for any nodes n_i and n_{i+1}, where n_{i+1} is the successor of n_i on the optimal path P. Since n_i and n_{i+1} are on an optimal path

$$g^*(n_{i+1}) = g^*(n_i) + c(n_i, n_{i+1}) \qquad (i = 1,\ldots,k-1)$$

therefore

$$g^*(n_i) + h(n_i) \le g^*(n_{i+1}) + h(n_{i+1}) \qquad (i = 1,\ldots,k-1)$$

By transitivity of i, we find that

$$g^*(n_i) + h(n_i) \le g^*(n_j) + h(n_j) \quad (i<j,\ i,j = 0,\ \ldots\ k). \blacksquare$$

Lemma 2. At any time, before the algorithm A^c terminates, there exists a node n in OPEN that is on an optimal path from the start node s to the goal set T with $f(n) \leq f^*(n)$.

We know that, there always exists a node n in OPEN that is on an optimal path from s to a goal node t before termination [Nilsson, 82]. Let the ordered sequence $(s=n_0, n_1, \ldots, n_k=t)$ be this optimal path, and we assume the node n is the first node in this sequence that is in the OPEN. (There must be at least one such node because s is in OPEN at the beginning and if t is removed from OPEN the algorithm has already terminated.) Obviously $g(n) = g^*(n)$, because all the ancestors of n on this optimal path are expanded.

Using this equation and lemma 1, we have that

$$f(n) = g(n) + h(n) = g^*(n) + h(n) \leq g^*(t) + h(t) =$$
$$= g^*(t) = f^*(s). \blacksquare$$

Theorem. The algorithm A^c is admissible.

We know the algorithm A terminates by finding a goal node if there is a path from the start node s to any goal node. Next we are going to show that the algorithm A^c is guaranteed to find an optimal path from s to a goal node.

Suppose the algorithm A^c were to terminate at some goal node $\bar{t}$ without finding an optimal path, i.e. $f(\bar{t}) > f^*(s)$. But, by the lemma 2, there existed a node n in OPEN just before the termination and on an optimal path from s to a goal node t with $f(n) \leq f^*(s)$. Thus, at this stage, the allgorithm A^c would have selected n for expansion rather than $\bar{t}$, contradicting our supposition that the algorithm A^c was terminated. $\blacksquare$

All the properties of the monotone restriction algorithm A^* proved by Nilsson [Nilsson, 82] can be preserved on the algorithm A^C, since those proofs do not exploit the fact that the algorithm is in the class of the algorithms A^*. The most important property is the following: The algorithm A^C has already found an optimal path to any node selected for expansion, i.e. if the algorithm selects n for expansion, $g(n) = g^*(n)$.

4. RELATIONSHIP BETWEEN THE ALGORITHM A^C AND ALGORITHM A^*

Now we have two admissible classes of algorithms: the algorithms A^C and the algorithm A^*. Both of them are in the class of the algorithms A. Let us examine their relationship.

We should like to show that class of the algorithms A^* encompasses the class of the algorithms A^C, i.e. if the heuristic function h satisfies the monotone restriction and $h(t) = 0$ for each goal node t then h is lower bound on h^* (i.e. $h(n) \leq h^*(n)$ for each node n).

If n is a goal node, we have $h(n) = 0$ and $h^*(n) = 0$ for definitions, therefore $h(n) \leq h^*(n)$.

If n is not a goal node and there does not exist a path between n and any goal node, we have $h^*(n) = \infty$ for definitions, so causing $h(n) \leq h^*(n)$.

If n is not a goal node and there exists a path from n to goal nodes, the proof is the following.

We assume that the sequence $(n=n_0, n_1, \ldots, n_k=t)$ is an optimal path from n to the closest goal node t. Using the monotone restriction we find that

$$h(n) - h(n_1) \leq c(n, n_1)$$
$$h(n_1) - h(n_2) \leq c(n_1, n_2)$$
$$\vdots$$
$$h(n_{k-1}) - h(t) \leq c(n_{k-1}, t)$$

The sum of these inequities is

$$h(n) - h(t) \leq \sum_{i=1}^{k} c(n_{i-1}, n_i).$$

Since $h(t) = 0$ and $h^*(n) = \sum_{i=1}^{k} c(n_{i-1}, n_i)$ for definitions, therefore we obtain

$$h(n) \leq h^*(n).$$

5. CONCLUSION

We saw that the class of the algorithms A^c is identical with the monotone restriction algorithms A^* by Nilsson. Henceforth it is enough to show that a heuristic function h satisfies the monotone restriction and $h(t) = 0$ for each goal t to get an admissible algorithm. It is easier to find a heuristic function of this kind than heuristics satisfying that h is lower bound on h^*, generally not known.

REFERENCES

[Fekete, 88] Fekete, I., Gregorics, T., Varga, L. Zs.:
Corrections to Graph-searching Algorithms
Fourth Conference of Program Designers,
ELTE, Budapest, 1988.

[Handbook, 82] Barr, A., Feigenbaum, E. A.:
The Handbook of Artificial Intelligence I.
HeurisTECH Press, Stanford, 1982.

[Hart, 68] Hart, P., Nilsson, N. J., Raphael, B.:
A Formal Basis for the Heuristic Determination
of Minimum Cost Paths
IEEE Trans. System, Man and Cybernet, 4, 1968.

[Nilsson, 82] Nilsson, N. J.:
Principles of Artificial Intelligence
Springer-Verlag, 1982.

Connection between AND/OR graphs and simple directed graphs

Sára Nagy

Eötvös Loránd University
Department of General Computer Science
Budapest

Abstract

In general the problem space of a problem-reduction representation can be modelled as an AND/OR graph. The AND/OR graph describes well the substitution of a problem with its subproblems. However the search procedures in the AND/OR graphs are more complicated than in the simple directed graph. Therefore our interest is to observe the procedures wich produce an equivalent directed graph out of an AND/OR graph.

This paper describes precisely the way of this transformation. It also examines cases when costs are given to arcs of the AND/OR graph. It presents how to find a solution in an AND/OR graph out of the solution of a directed graph.

1. Introduction

A problem can also be solved by attempting to reduce it to its components. The subproblems are simpler than the original problem. The received subproblems can be reduced in further components. The reduction is finished if we have got only primitive and/or unsolvable problems. We say that a problem is primitive if we know its solution. We say that a problem is unsolvable if we know from somewhere that no solution of the problem exists or we don't know the solution of the problem and we are unable to reduce it.

AND/OR graphs can be used as a model of the problem space of problem-reduction representations. In a problem-reduction representation the original problem / and recursively each subproblem / is not only divided into subproblems but it can be substituted by other problems.

In AND/OR graphs the problems are associated with nodes. If all of the subproblems must be solved in order for the original problem to be solved, the nodes which represent these subproblems are AND successors /they are denoted in our illustrations by a curved line joining the arcs/. Otherwise the problems are OR successors of the original problem.

In the course of our examination we make an effort to avoid the mixed ramifications. A mixed ramification can be eliminated with the initiation of new nodes.

2. Production of directed graphs out of AND/OR graphs

There are many search strategies for AND/OR graphs, but they can be put into practice with difficulty. For this reason it is interesting how we follow the search in an AND/OR graph with the aid of a directed graph. In reality we give a directed graph model of the problem space of problem-reduction representation instead of AND/OR graph model. Although the AND/OR graph model of the problem space is more clearly arranged than the directed graph model, the latter can be put to better use because here the well-known search strategies can be used.

Let's see how an AND/OR graph is formulated as a directed graph. Suppose that the all nodes of the AND/OR graph have a label. Now we describe how to produce a directed graph from an AND/OR graph:

1. Attach a label to the root node of the directed graph. In the directed graph all the labels make a set of labels. The set of the labels of the root includes only the label of the root node of the AND/OR graph. Put the root node on a list called WORK-UP.

2. If WORK-UP is empty then stop.

3. If WORK-UP isn't empty then select the first node on the list and omit it. Call this node n. Assume that the set of labels of n is equal to $\{n_1, \ldots, n_k\}$ / n_i is a label in AND/OR graph/.

4. Repeat the followings k times:

4.a If n_i has OR successors then install successors of the same quantity of n in directed graph as the number of OR successors. A successor is labeled by replaceing n_i with its corresponding successor in the set of labels.

4.b If n_i has AND successors then install only one successor of n in directed graph and replace n_i with the set of labels of all its successors.
/Having got a node already exiting in a directed graph then it isn't necessary to install a new node only a new arc./

5. Put the new nodes on WORK-UP.

6. Go to 2.

Figure 1 illustrates a problem space by AND/OR graph. The primitive problems in the graph are indicated by solid circles and the unsolvable problems by empty circles.

Figure 2 shows the directed graph corresponding to AND/OR graph in Figure 1. This directed graph is modeling the same problem space as the AND/OR graph. In the directed graph we framed the terminal nodes. We say that a node is terminal in this directed graph if all members of its label set denote primitiv problems.

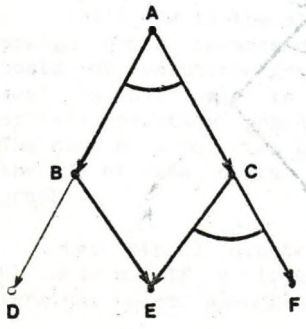


Figure 1.

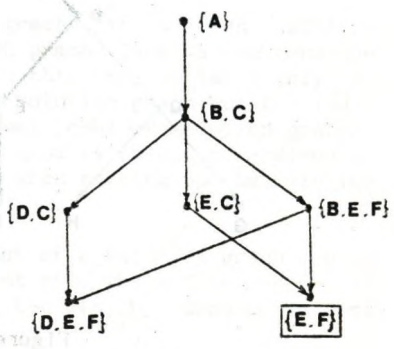


Figure 2.

Observe that our algorithm defines too big state space because it contains too much essentially equivalent paths. For example in Figure 2 two paths are directed to node E,F . These paths aren't essentially different because ones we dealt first node C and then node B and next we dealt first node B and then node C.

Thus the step 4 of the algorithm must be modified:

4'. Select a label with label set of node n and call this label n_i.

The 4.a and 4.b correspond with the afore-mentioned.

When we select a n then it is advisable to give preference to nodes with AND successors because thus the number of nodes in a directed graph there will be few nodes than in other case / see Figure 4.a and 4.b /. If we transformed the AND/OR graph in Figure 3 original algorithm, we would have a graph with 42 nodes. It can be seen in Figure 4.b that only 16 nodes are sufficient.

Since any AND/OR graph can be formuleted as a directed graph, it is easy . to see that an algorithm for searching AND/OR graphs which is structurally the same as the one for searching directed graphs can be defined. The algorithm can be defined by selecting a node to expand and a label as well from its label-set. Then we must expand it according to 4.a or 4.b. The algorithm is terminated if it reaches a node with a label-set containing only primitive problems. If there is an unsolvable node in the label-set of any node, it isn't worth working with it.

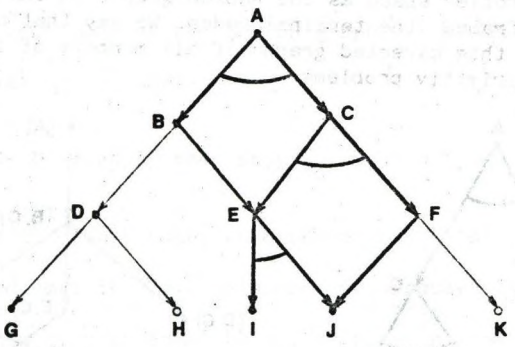


Figure 3.

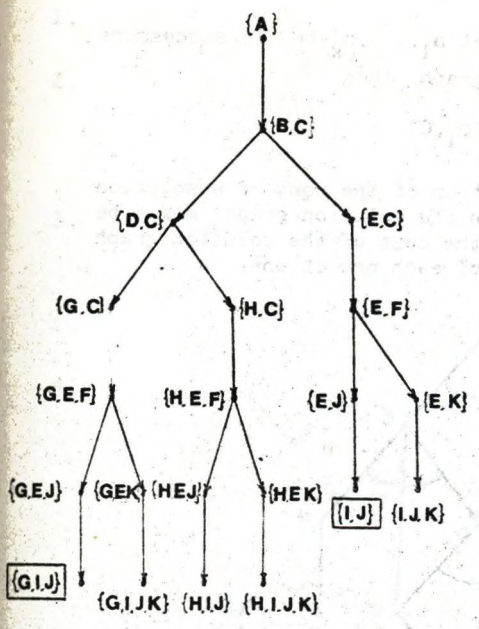


Figure 4.a

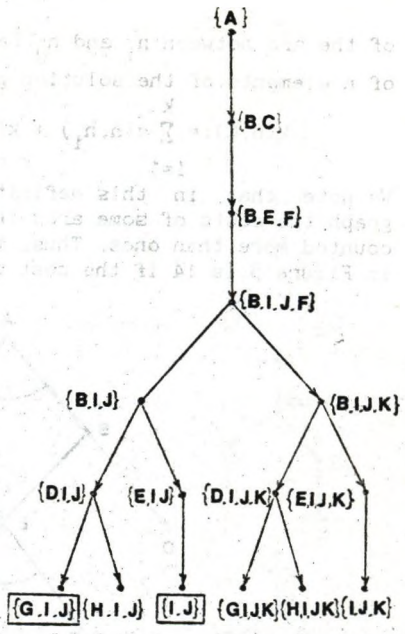


Figure 4.b

A solution subgraph has been genereted for AND/OR graph by directed graph by considering solved nodes corresponding to the label-set of the terminal node in directed graph. Then a node is solved if it has OR successors at least one of which is solved or it has AND successors all of which are solved. / This definition assumes that our AND/OR graphs contain no cycles./ In Figure 1, 3 and 5 the derkened arcs indicate a solution graph for those AND/OR graph.

3. The cost of solution graphs

Analogoue to the directed graphs, it is often useful to assign costs to arcs of AND/OR graph. Thus we can consider costs of solution graphs. In this case it isn't only the goal is not only to find a solution graph but to find a optimal solution graph /minimal cost of solution graph/. The cost of a solution graph is usually taken determined by the sum of the costs of the arcs makeing up the solution graph.

Let k(n,C) denote the cost of a solution graph rooted at node n. If n is an element of C where C is the set of terminal nodes, then k(n,C):=0. Let $c(n_i,n_j)$ denote the cost

of the arc between n_i and n_j. Let $n_1, \ldots, n_k$ be the successors of n elements of the solution graph. Then

$$k(n,C) := \sum_{i=1}^{k} c(n,n_i) + k(n_i,C).$$

We note that in this definition of the cost of a solution graph the costs of some arcs in the solution graph might be counted more than ones. Thus, the cost of the solution graph in Figure 5 is 14 if the cost of each arc is one.

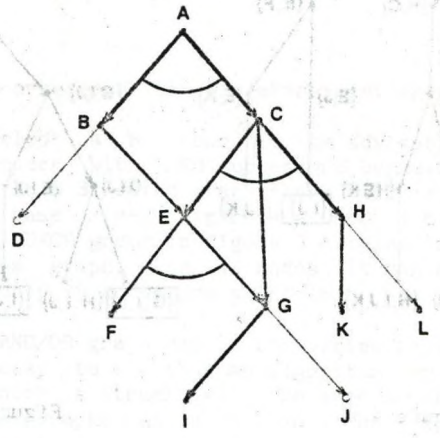


Figure 5.

If we want to take also the cost into add up then we must put down the nodes of AND/OR graph by multiplying then into the label-set. In the directed graph the cost of an arc is equal to the cost of arc in AND/OR graph if it is directed to an OR successor, or the sum of costs of arcs directed to AND successors. Figure 6 illustrets the search outlined above.

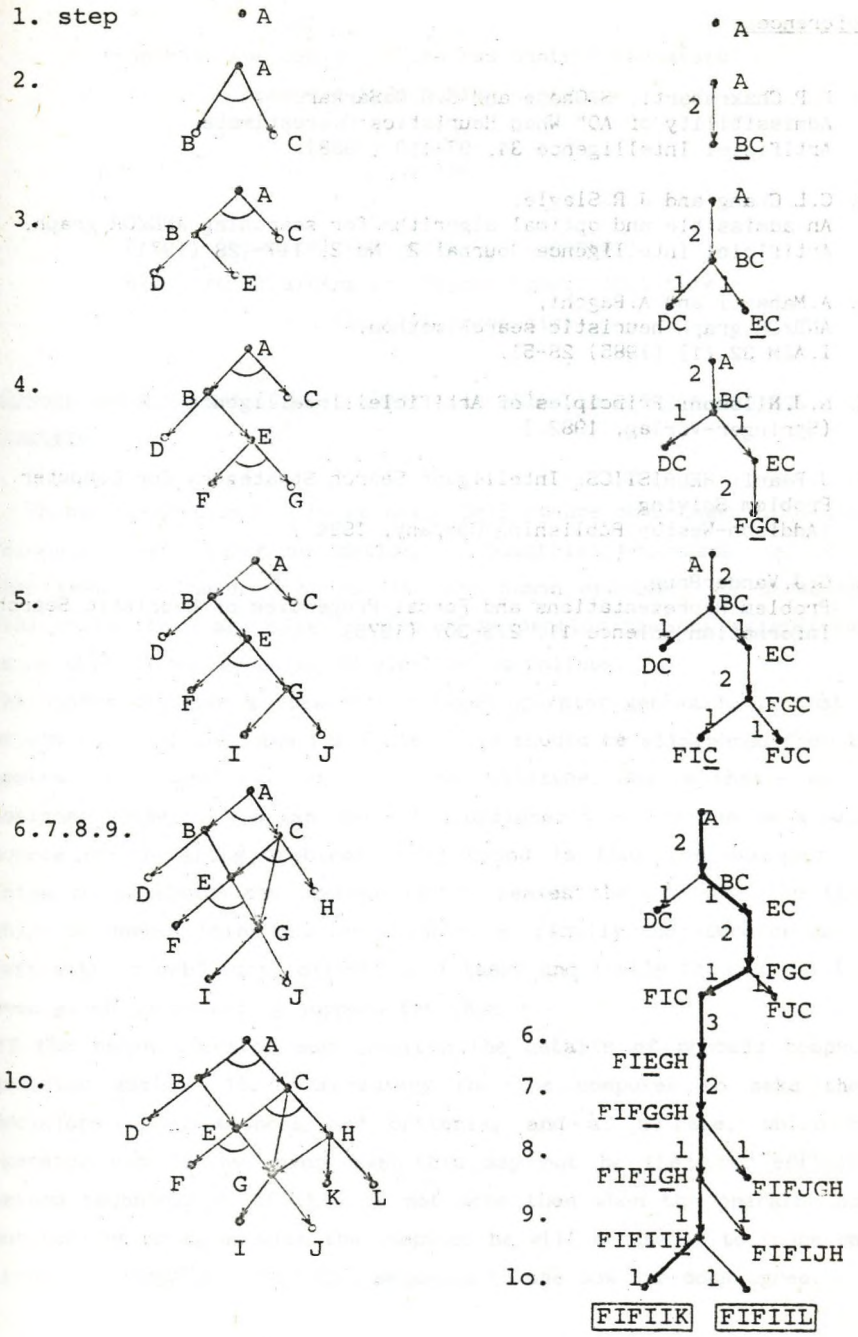

1. step
A
2.
B
C
2
BC
3.
D
E
1
DC
EC
4.
F
G
FGC
5.
I
J
FIC
FJC
6.7.8.9.
H
FIEGH
FIFGGH
FIFIGH
FIFJGH
FIFIIH
FIFIJH
lo.
K
L
FIFIIK
FIFIIL

Figure 6.

References

1. P.P.Chakrabarti, S.Ghose and S.C.DeSarkar,
   Admissibility of AO* When Heuristics Overestimate,
   Artificiel Intelligence 34, 97-113 (1988).

2. C.L.Chang and J.R.Slagle,
   An admissible and optimal algorithm for searching AND/OR graph,
   Artificiel Intelligence Journal 2, No.2, 117-128 (1971).

3. A.Mahanti and A.Bagchi,
   AND/OR graph heuristic search method,
   I.ACM 32 (1) (1985) 28-51.

4. N.J.Nilsson: Principles of Artificiel Intelligence
   (Springer-Verlag, 1982.)

5. J.Pearl: HEURISTICS: Intelligent Search Strategies for Computer
   Problem Solving
   (Addison-Wesley Publishing Company, 1984.)

6. G.J.VanderBrug,
   Problem Representations and Formal Properties of Heuristic Search
   Information Science 11, 279-307 (1976).

Proposals for design of process control operators' computer information systems

Lajos Izsó

Technical University of Budapest
Teachers' Training and Psychological Institute
Department of Ergonomics

1. Common usage of man-computer task allocation in process control systems

Though theoretically the relative performance advantages of men and machines are well known automation of industrial processes may expand rather than eliminate problems with the human operator. As BAINBRIDGE (1983) wrote there are some ironies of automation the most significant ones of which from our point of view are as follows.

- The system designer's view of the human operator generally is that he or she is unreliable and inefficient, so should be eliminated from the system. There are two ironies of this attitude. One is that - as the designer himself is a man too - the designer's errors can be a major source of operating problems. The second is that the designer who tries to eliminate the operator still leaves the operator the tasks which he cannot think how to automate. So finally the operator can be left with an arbitrary collection of tasks and little thought may have been given to providing support for them.
- If the human operator must monitor the details of process computer decision making, it is necessary for the computer to make these decisions using methods and criteria, and at a rate, which the operator can follow, even when this may not be the most efficient method technically. If this is not done then when the operator does not believe or agree with the computer he will be unable to trace back through the system's decision sequence to see how far does agree.

- If because of his or her bad previous experiences the operator doesn't trust the computer, the use of computer increases the operator's strain instead of decreasing it. EPHRATH (1980) - cites BAINBRIDGE (1983) - has reported a study in which system performance was worse with computer aiding, because the operator made the decisions anyway, and checking the computer added to his workload.

2. The role of operator's cognitive model on the system to be controlled in the effectiveness of the operator's activity

In a process control system there are two general kinds of task left for an operator. The operator may be expected to monitor that the system is operating correctly. and if it is not he or she may be expected to take over and stabilize the process. To take over requires manual skills, to diagnose the fault as a basis for shut down or recovery requires cognitive skills. For the designer of the computer information system the cognitive skills are of great importance. An operator will only be able to generate successful new strategies for unusual situations if he or she has an adequate knowledge of the whole system, including first of all the process to be controlled itself. The significance of the operator's cognitive modell in the effectiveness has been many times verified - e. g. HALPIN at al (1973), BRIGHAM and LAIOS (1975), LANDEWEERD (1979), LANDEWEERD at al (1981), ANTALOVITS (1985), (1986) - now the main problem is how to design computer information systems which are in accordance with the operators' basic knowledge on the process and at the same time can help the operator to build up more and more adequate and of higher and higher fidelity cognitive models. Concerning operators' basic knowledge there are different interface design recommendations depending on wheter the operator is naive, novice, competent or expert (CHAFIN(1981)).

In contrast with tasks such as database searching, text editing and programming which are relatively static in the sense that they are paced by the human's decision, the industrial processes are dynamic systems, as their future outputs depend on their past outputs, as well as on inputs generated by humans, computers, and the enviroment. That is why

the time factor has a particular importance in the cognitive model: "the system does not wait for the human or computer to make decisions" -ROUSE (1981). As the system "does not wait" the process may get out of the control and it may lead to damages or even to catastrophe.

For preventation of such faults there is a need for an effective process computer alarm system. For the alarm system extraordinarily strict requirement all the displayed informations to be consistent with process model and to be self-consistent as well. As an example, it is known, that in 1979 the nuclear reactor at No. 2 Station, Three Mile Island, Pennsylvania, suffered an accident sequence which resulted in severe core damage and in some radioactive release. The operators experienced problems in diagnosing the faults - LEES (1983) - and that the failure to diagnose correctly was a major factor in the accident. The operators failed for over 2 hr to recognize that a pilot-operated relief valve, which had supposedly opened and then closed, had in fact remained stuck open. The operators failed to realise too that under the abnormal conditions prevailing, the indication of water level in the pressuriser vessel was not a true indication of water content in the reactor vessel. In both cases there were several pieces of information which were true indicators of the system state but also some information which appeared to be inconsistent with this state. Judging by data available it seems very probable that, at least partly, similar causes led to accidents in Flixborough and in Chernobil too.

An important role of computer information system can be interpreted on the basis of cognitive psychology. It is known that the span of Short Term Memory (STM) equals 7±2 "units", but these "units" may be very simple (e. g. letters), of medicum complexity (e. g. syllables or words) or complex (e. g. common sayings or poem-details). MÉRŐ (1988) defines these "units" as "patterns" and in accordance with experiences he postulates that a "pattern" can be stored in STM only if this pattern already has a representation in Long Term Memory (LTM). From this for the designers of information systems it follows that they must design the systems so that these systems should be able:

- to inform the operator about the state of the process in accordance with his or her actual cognitive model (e. g. by displaying data

really relevant, by using technical terms operators accustomed to etc.)

- to provide informations in a way suitable to form more and more complex "patterns" of the parameters characterising the process
- to be easily modified by the operator if his or her more complex "patterns" make it imperative.

3. Experiences gained in the Szolnok Paper Mills in the course of installation of an up-to-date computerized process control system

In 1982 began the installation of the new computerized paper manufacturing line in the Szolnok Paper Mills the total cost of wich was over 62 millions US$ s. We took part in the personnel selection and in the period 1982-86 we had been also following with attention the in-service suitability of the 25 process control operators chosen from among 141 nominees. Here are summarized only the main experiences relevant to information system design, for further details see IZSÓ (1988/1), (1988/2).

- The computerized process control activity requires quite different cognitive skills than the manual control.

1. The new system began to manufacture in January 1984 by manual control till April and since then by computer control. The effectiveness of the control measuring by the total break-down time in January, Februery, March and April did depend on the shift (group of operators) while an May, June, and July did not. After a year, as new cognitive skills had been developed, differences occur again but in a changed order of succession.
2. The correlation coefficiens between the effectiveness of the control and the effectiveness of a computer-simulated simple process control task were as follows:

   in 1982 r=0,230 (n=11, n.s.), working on the old paper-line whith no computer aid

   in 1984 r=0,359 (n=15, n.s.), working on the new paper-line, since May with computer control, but still not in a skilled way

in 1986 r=0,567 (n=16, p<0,05), working on the new paper-line in the consolidated system with computer control and in skilled way.

- Without user-friendly software interface and appropriate teaching courses the formation of more complex cognitive patterns goes on very slowly. Till the summer of 1985 some very informative display reports practically were not called by the operators on the VDUs because these were not compatible with their relatively undeveloped cognitive models. For example:

1. The trends, i.e. the graphic or tabular presentations of the progress of paper parameters in time were permanently disregarded.
2. The histograms, i.e. the presentations of distribution functions graphs of paper parameters were also disregarded. The main problem probably was the mathematical formulation, since such quantities as "2 sigma" had no meaning for the operators at all.
3. Roller analysis, i.e. informations about the quantity of paper already rolled up at the end of the paper-line also was never called. This informations would be very important for the workers of finishing-room, but they did not know this possibility, the process operators however were not interested in it.
4. Digital data display, i.e. displaying any measured data available for the computer was also rarely used, because for using it the operator ought to know the concrete memory addresses of the data.

4. Some general proposals for design of process control operators' computer information systems

The computer information systems should

- have a clear, relatively simple and fixed in written form process-model to be taught in the preparation courses which is at the same time flexible enough to provide a basis for further in-service development of this model
- have a simple model of data collecting and information system too also to be tought

- provide all the informations in accordance with this basic cognitive model
- give possibility for the operators to design (to choose, to combine and to reprogramme) their screen formats, graphic or tabular presentations and data to be displayed within limits
- be able to simulate typical process-disturbances by the help of mathematical model and also to reproduce events already taken place for training of process operators thus providing then possibility to form more and more complex "patterns".

References

1. ANTALOVITS, M. 1985. Információ feldolgozás az operátori tevékenységben (kandidátusi értekezés), Budapest
2. ANTALOVITS, M. 1986. Information processing in the activity of operators (abstract of Ph. D. Thesis), Budapest
3. BAINBRIDGE, L. 1983. Ironies of Automation, Automatica, Vol. 19. No. 6. pp. 775-779.
4. BRIGHAM, F. R., LAJOS, L. 1975. Operator performance in the control of a laboratory process plant. Ergonomics, 18, 53-66.
5. CHAFIN, R. L. 1981. A model for the control mode man-computer interface. Proc. 17 th Ann. Conf. on Manual Control, UCLA. JPL Publication 81-95, pp. 669-882.
6. EPHRATH, A. R. 1980, Verbal presentation. NATO Symposium on Human Detection and Diagnosis of System Failures, Roskilde, Denmark.
7. HALPIN, S. M., JOHNSON, E. M., THORNBERRY, J. A. 1973, Cognitive Reliability in Manned Systems. IEEE Transaction on Reliability, Vol. R-22, No. 3., 165-169.
8. IZSÓ, L. 1988/1, Az emberei megbizhatóságot meghatározó hardver-, szoftver- és feladatjellemzők ember-számítógép rendszerekben. (Kutatási jelentés.) Kandó Kálmán Villamosipari Müszaki Föiskola, Budapest
9. IZSÓ, L. 1988/2, Factors influencing human reliability and strain in man-computer systems. (Research report). Thecnical University of Budapest, Department of Ergonomics, Budapest

10. LANDEWEERD, J. A. 1979, Internal representation of a process, fault diagnosis and fault correction. Ergonomics, 22, 1343-1151.
11. LANDEWEERD, J. A., SEEGERS, H. I.J.L., PRAAGMAN, J. 1981, Effects of instruction, visual imagery and educational background on process control performance. Ergonomics, 24, 133-141.
12. LEES, F. P. 1983, Review paper. Process computer alarm and disturbance analysis: review of the state of the art. Computers and Chemical Endineering, Vol. 7, No. 6. pp. 669-694.
13. MÉRŐ, L. 1988. A mesterséges intelligencia és a kognitiv pszichológia kapcsolata (tanulmány), Eötvös Loránd Tudományegyetem BTK Kisérleti Pszichológiai Tanszék, Budapest
14. ROUSE, W. B. 1981, Human-Computer Interaction in the Control of Dynamic Systems, Computing Surveys, Vol. 13, No 1. 71-99.
15. SINGLETON, W. T. 1974, Man-Machine Systems, Penguin Modern Psychology, Cox Wyman Ltd. London

CONNECTIONS BETWEEN AI AND COGNITIVE PSYCHOLOGY

László Mérő
Loránd Eötvös University
Department of Experimental Psychology

Abstract

This paper briefly examines four major points in the connections between AI and cognitive psychology. First, some results of cognitive psyshology suggest that there may be significant principial differences between human information processing and the commonly used frameworks of AI. Our second point is that nevertheless, the research motives of these disciplines are common in many aspects. Third, the import of some results in cognitive psychology to AI are considered, e.g. what volume of cognitive schemata should be incorporated into an intelligent system? Fourth, the import of some results in AI to cognitive psychology are also investigated: the changing of our notions about intelligence and creativity in the light of the AI results.

I. Should AI find its foundations in psychology?

The inventors of most of the commonly used AI techniques, like frames, semantic networks, repertory grid analysis, etc. try to find some psychological backgeound to support the proposed architecture. This kind of efforts shows that AI researchers are not totally satisfied with their theoretical foundations: the purely mathematical and/or engineering methods did not result in really intelligent programs. AI, as an engineering/mathematical discipline should not necessarily be interested in alternative working ways of the same performance, i.e. in natural intelligence. Nevertheless, AI is definitely interested in our knowledge about human intelligence because it outperforms all of today's AI products. This interest is, however rather selective: AI researchers take gladly cognizance of psychological models and results that support their way of thinking while more or less neglects those results that contradict it.

Cognitive psychologists are strongly interested in the shortcomings of our thinking, as the subjective probabilistic judgements that are sometimes very false. Let us show just one example. Suppose there is a city where only two colors of cars exist: green and blue. 85% of the cars are blu, the other 15% are green. Once upon a time a driver failed to stop after road accident. However, there was a witness who asserted that the criminal car was green. Psychologists have investigated the perceptive capabilities

of the witness and realized that his judgement of color at the given speed is correct only in 80% of the cases. The question arises: what is now the probability that the criminal car was really green?

Most of the people give a radically wrong estimate on this question. A very frequent answer is 80%, but very few people say less than 50%. On the other hand, this problem is clearly a Bayesian one, and it is easy to compute using the Bayes theorem, that the correct probability is 41%. When we are looking to the problem in another way, this result might be highly intuitive: the witness increased the probability of being the car green from the initial 15% to a 41% a posteori probability - quite a nice increasement. However, our normal intuition looks to the problem from quite a different aspect and the result is surprizing for most of us. This example is not the least singular: similar kinds of results abound in cognitive psychology. The referred literature contains a great deal of fallacies in our everyday thinking. Even well-trained professional scientists are prone to many kinds of severe errors when their probability estimates and inferring mechanisms are challanged.

Now a very hard question arises: should an AI system follow the shortcoming of the human estimate or not? If we say yes, the performance of our AI system will suffer. If we say no, the original sin of abandoning the way of human intelligence will have been committed. As an engineer, I must suggest to keep using the Bayesian way (or other

normative principles like the Dempster rule or some kind of fuzzy logic), because they work quite well. On the other hand, as a cognitive psychologist I must look for models that incorporate these kinds of fallacies even if the performance of the resulting models is not optimal. Our present understanding about the human mental processes does not afford building really human-like models of thinking.

II. Where does the meaning reside?

In a summary manner, a general moral of the lessons of numerous experiments in cognitive psychology sounds as follows: the meaning resides everywhere, at every level of human information processing. We shall illustrate the general style of cognitive psychology experiments leading to this moral by just one example. Suppose we flash a letter R or a letter M on the display of the computer. The subject has to push the button at his/her right hand if he/she perceives a letter R, and push the other button if a letter M is perceived. The time of flashing the letter is determined so that the ratio of correct responses be 60%. (This means that the subject must perceive a certain small amount of information but not enough for a sure decision). After that all the circumstances remain the same but the display image changes: either the word MAKE or the word RAKE will be flashed. The subject has to tell the first letter of the word seen. Usually the ratio of correct answers significantly increases in the second case. If a meaningless

word is displayed, the hit ratio will not increase.

Perhaps the most important common point in AI and cognitive psychology researches is the effort to allocate the point where the meaning appears. Both disciplines try to attack this problem, and the results converge in a nice however may be distressing way. Cognitive psychology experiments demonstrate that the meaning resides largely everywhere, at every point of our thinking. AI results demonstrate that the meaning does not reside at any given point of the AI systems: neither at the links of the semantic networks, nor in the sophisticated frame structures, even not in any kinds of problem solving heuristics. As Lenat and Brown put it when analysing why the lovely and efficient AM and EURISKO programs appear to work: "although the system in principle contains a complete characterization of what the operators of the language mean (the system has enbedded within itself a representation of EVAL - a representation that is, in principle, modifiable by the system itself) the system nevertheless contains no theory as to what the data structures denote. Rather, we (the human observers) attribute meaning to those structures."

III. The import of cognitive psychology to AI

The concept of cognitive schema in cognitive psychology is much subtler than any of the analogous concepts in AI. A cognitive schema is some meaningful thing that is changing

all the time. It plays an active role in all phases of human thinking, from perception to memorizing and problem solving. As this concept is that very soft, even somewhat vague, it is very hard to tell whether such entities really exist in our brain or not. Nevertheless, this concept has helped in designing many experiments that led to a much deeper understanding of human thinking mechanisms. On the other hand, the concept of the cognitive schema is highly intuitive, we cannot explicate it in exact mathematical terms. Therefore this concept presently cannot be directly modelled in AI programs.

However, cognitive psychology was able to give even some quantitative estimates about the number of cognitive schemata, whatever they be. It has been shown for example, that a grandmaster of the most professions masters some ten thousands of cognitive schemata in his/her field. This result can explain the apparent shortcomings of today's AI programs: none of them contains that much sophisticated cognitive entities. A cognitive schema is not just a memory unit: the number of distinct memory units can be estimated to 2-3 orders of magnitude larger than the number of cognitive schemata. It is possible that meaning resides simply in the complexity that can be afforded by the human brain, and this complexity perhaps can be measured by the quantity of interrelated cognitive schemata that change each other all the time.

The great challange to both AI and cognitive psychology is to find the way how the several millions of memory units

are organized into some ten thousands of schemata in our brain. This organizing principle may be largely different from the commonly used computer architectures. Cognitive psychology has clearly demonstrated some phenomena that are presently inexplicable in computer terms. A simple example is the phenomenon of the capacity limits of the short term memory. It is not clear why should a computer model have a construct like short term memory at all, and once it has one, why should it be so strongly limited if no engineering reasons support this limitation. This way cognitive psychology offers some nice puzzles to solve for AI, and the solution of these problems may enhance our understanding both about the proper ways of building intelligent systems and about the working principles of human intelligence.

IV. The import of AI to cognitive psychology

The basic objectives of cognitive psychology has been to find well-working models of human cognition and thinking. The initial demonstrative successes of AI have also promised a great breakthrough for cognitive psychology by offering external, normative working models that should only be refined to reflect some funny human features. However, in the last 30 years the initial enthusiasm has dissolved, because AI models, as we have seen, are presently not able to catch the major features of human thinking. To put it cynically, from a cognitive point of view: if an AI model is working reasonably well, it cannot be considered as

intelligent, and if an AI model can perhaps be considered as intelligent, it is not working well.

Less cynically, the results of AI have continuously and radically changed our notions about intelligence. Expert systems or chess programs are definitely working quite well, but their working mechanisms radically differ from the human thinking ways. On the other hand, sophisticated knot-like structures, like Winograd's SHRDLU may show some similar features to human thinking, but their performance seems to be very limited, their working principle could not yet be generalized to broader domains than toy blocks, etc.

A still more meaningful lesson of AI for cognitive psychology can be found in understanding the notion of creativity. Chess programs, expert systems or especially EURISKO produce unequivocally creative ideas without being intelligent in the human sense. As Schank and Oehn put it, the AI experience has thought us that creativity may be just an extreme case of intelligence, a special kind of memory organization that is generally characteristic to those people who are usually considered intelligent by other people. The apparent creativity of AI programs totally lacking any real intelligence may help us to dissolve the myth of creativity.

Bibliography

To Section I.

Cherniak, C. (1986). Minimal Rationality. The MIT Press.

Einhorn, H. J., Hogarth, R. M. (1986). Judging probable cause. Psychological Bullentin 99. 3-19.

Goldman, A. I. (1986). Epistemology and Cognition. Harvard University Press.

Johnson-Laird, P. N. (1983). Mental models. Cambridge University Press.

Kahneman, D., Slovic, P., Tversky, A. (ed., 1982). Judgement under uncertainity: Heuristics and biases. Cambridge

Nisbett, R., Ross, L. (1980). Human Inference. Prentice-Hall.

Rips, L. (1983). Cognitive processes in reasoning. Psychological Review 90. 38-71.

Zajonc, R. B. (1980). Feeling and thinking: Preferences need no inferences. American Psychologist 35. 151-175.

To Section II.

Anderson, J. R. (1984). Cognitive Psychology. Artificial Intelligence 23. 1-11.

Lenat, D. B., Brown, J. S. (1984). Why AM and EURISKO appear to work. Artificial Intelligence 25. 269-294.

Mérő L. (1984). A heuristic search algorithm with modifiable estimate. Artificial intelligence 23. 13-27.

Searle, J. (1980). Minds, brains and programs. The Behavioral and Brain Sciences. 3. 417-457.

Standing, L. (1973). Learning 10,000 pictures. Quarterly

Journal of Experimental Psychology. 25. 207-222.

Wheeler, D. D. (1970). Process in word recognition. Cognitive Psychology 1. 59-85.

To Section III.

Cermak, L. S., Craik, F. I. M. (ed., 1979). Levels of processing in human memory. Erlbaum.

Gardner, H. (1985). Frames of mind. Heinemann, London.

Landauer, T. K. (1986). How much do people remember? Some estimates of the quantity of learned information in long term memory. Cognitive Science 10. 477-493.

Mérő L. (1984). Heurisztikus eljárások a mesterséges intelligenciában. Pszichológia 4. 241-259.

Schank, R. C. (1979). Interestingness: Controlling inferences. Artificial Intelligence 18. 273-297.

To Section IV.

Cherniak, C. (1988). Undebuggability and computer science. Communications of the ACM 31. 402-412.

Dreyfus, H. L., Dreyfus, S. E. (1986). Mind over machine. The Free Press.

Hidi, S., Baird, W. (1986). Interestingness - A neglected variable in discourse processing. Cognitive Sciences 10. 179-194.

Sternberg, R. J. (1982). Handbook of human intelligence. Cambridge University Press.

Vernon, P. E. (1970). Creativity. Penguin Modern Psychology Readings.

Winograd, T. (1980). Language as a cognitive process. Addison-Wesley.