tanulmányok

117/1981

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

PRINCIPLES OF COMPUTER AIDED SYSTEM DESCRIPTION

Irta:

Knuth Előd, Radó Péter

Tanulmányok 117/1981.

A kiadásért felelős:
DR VÁMOS TIBOR

ISBN 963 3111 114 5
ISSN 0324-2951

Elöd Knuth, Péter Radó

PRINCIPLES OF COMPUTER AIDED SYSTEM DESCRIPTION

November 1980 II/13

ABSTRACT

The most fundamental general models applicable for system description, design, analysis, and documentation purposes are surveyed in the paper and proposed description methodologies for all of them are given. Models are divided into three levels namely the "conceptual", "logical", and "machine-oriented" ones.

The paper is based on the terminology of system SDLA (see ref.) but the reader need not know it exactly. Another aspects of computer aided system design different from the description process can be found in subsequent papers.

This work is part of the SDLA project supported by the Hungarian National Bureau of Computer Applications and by the Hungarian Academy of Sciences.

CONTENT :

0. INTRODUCTION

This paper deals with computer aided system descriptions (specifications) which can be put into a database and be used as a basis for system analysis and documentation purposes. To perceive the proper situation however, first we must distinguish between:

0.1 Formatted versus formal specifications

The so called entirely formal specification methods (like VDM and LDM) provide full semantical descriptions as a sufficient basis for automatic system development. This is something different from our purposes and can be used in other circumstances and to meet other aims.

Formatted specifications treated in this paper are also strictly formal in their syntax but leaving the semantics in a degree free. This freedom meets the requirements of users not trained in mathematical logical specifications, and on the other side it is useful too when being at the beginnig stages of top-down specifications knowing nothing exactly.

We also deal, however, with semantical aspects provided by semantic-constraints and semantic-checking facilities (as it is introduced in the system SDLA) but this can be considered as only a "human-close" approach, and is better comparable to the efforts recently made in the field of semantic reflection capabilities of the databese relational model.

0.2 Computer aided system design

When we would like to study it in general we could deal with the subfields shown by table 0.1 .

I. General principles (horizontal view)

- Description methodologies
- Consistence checking methods
- Analysis facilities
- Documentation methodologies

II. Complete particular systems (vertical view)

(classified by application fields)

Table 0.1

The aim of this paper is looking over the most fundamental models used in various stages of computer aided system design and giving only the description methodologies to apply. Other topics shown in table 0.1 can be found in subsequent papers.

1. BASIC NOTIONS

We use the notations of SDLA (System Descriptor and Logical Analyzer - see references) throughout the paper but the reader need not be familiar with its details exactly. The formalism used is selfexplanatory, and their most important aspects can be understood from the following sections.

1.1 The two level description approach

We use a two level (concept definition + system description) approach similarly as the GENERALIZED ANALYZER (see ref.) but in a different way and manner (mainly of semantic nature).

At the definition level we define the framework of our system model to be used at the description level by giving concepts, relationship types, their main properties, and declaring permissible user language sentence forms.

During the phase of the descriptive dialogue everything defined previously can be used resulting in instanciations of the predefined concepts reflecting the particular model (system or phenomenon) described. These are put into a database and can be subject of system analysis, checking, and various documentation procedures.

1.2 The concept refinement mechanism

Suppose we want to use an object class called "file" during the descriptive dialogue what we characterize with just one attribute namely the block-size. Suppose we also need special file types e.g. the "printfile" having extra attributes as the page-size, and line-length. These can be defined in the following form:

```
(1.1)     concept file(blksize:integer);
          concept printfile is file(pagesize:integer,
                                    linelength:integer);
```

Now if e.g. "opening-procedure" is another concept referring to a "file" object i.e.

```
(1.2)     concept opening(subj:file);
```

then at the instanciation level not only file objects but any specialized versions (e.g. printfile objects) are also accepted as actual references.

In general, this can be formulated in the following way. When data instances are presented their attribute values must correspond in type to the attribute specification given at the meta level, however, for a specified type any of its subtypes is also accepted as a value. This principle is called the generalized type-coincidence law.

1.3 Form clauses

For every concept declared at the definition level any number of (absolute and relative) "form clauses" can be attached. These will constitute the user language to be applied during the descriptive phase. We illustrate it by an example. Suppose we wrote at the definition level:

```
(1.3)     concept process;
          form absolute: process;

          concept data;
          form absolute: data;

          concept use(user:process,used:data);
          form user: uses used;
          form used: used-by user;
```

Now e.g. the following structures are allowed to write at the description level:

```
(1.4)     P: process;
             uses data D;

          E: data
             used-by P, Q;

             etc.
```

2. GENERAL CONCEPTUAL MODELS

In the conceptual level of modelling only very simple structures and a low number of concepts are needed. We can not speak about generally adopted or advised methodologies in this level, one can choose or form his own terms at any time so as to get the most adequate model of his problem or of a real world phenomenon to be described.

Models in the conceptual level have, however, the common property that each of them have a kind of graph structure. There is a great diversity of different graph models, and we can just give a foretaste of them.

One may raise the objection that graph models suits much better to real graphic representations (ensuring better lucidity) than to descriptions as textual data. It is true, of course, however, there are two reasons making it necessary to desribe and store such simply structured models in databases, namely the extent of really practical models, and on the other side the analysis capabilities provided by general software tools.

2.1 Simple directed graphs

To start with taking a practical view, consider the conceptual model outlined in fig. 2.1 .

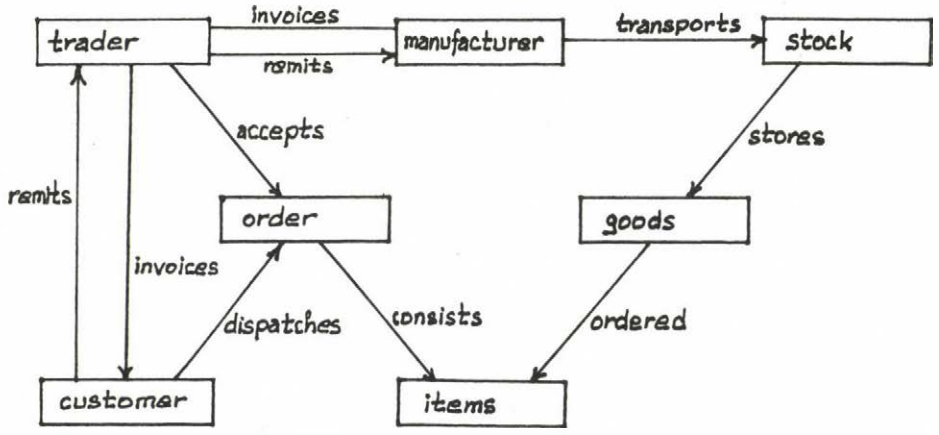


Fig. 2.1

To represent and describe structures as simple as above we need just two concepts e.g. like:

```
(2.1)      concept entity;
           concept relationship(of:entity,to:entity);
```

and, in order to achieve easily readable descriptive forms, we can supplement them by form declarations as follows

```
(2.2)      concept entity;
           form absolute: entity;

           concept relationship(of:entity,to:entity);
           form of: relation-to to;
           form to: relation-from of;
```

Now using these declarations we can describe the scheme expressed by fig. 2.1 as:

```
(2.3)      trader: entity;
              accept: relation-to order;
              invoice: relation-to customer;
              remit-m: relation-to manufacturer;

           customer: entity;
              dispatch: relation-to order;
              remit-c: relation-to trader;

           order: entity;
              consist-of: relation-to items;

           manufacturer: entity;
              invoice-m: relation-to trader;

           items: entity;

           stock: entity;
              store: relation-to goods;

           goods: entity;
              ordered: relation-to items;
```

Remarks:

1. Note that though during the decription we used the "forward" relation description form ("relation-to") only, however when regaining reports from the database both aspects appear on request e.g. the entity "customer" will look like as

```
           customer: entity;
              dispatch: relation-to order;
              remit-c: relation-to trader;
              invoice-c: relation-from trader;
```

2. We can see the model contained in fig. 2.1 is incomplete from several viewpoints. It does not reflect dynamic properties, time relationships, mutual ordering between actions, conditions of actions, etc. Nevertheless the information contained in the figure is, in fact, of basic importance at this level, those aspects still missing can be expressed on another levels later.

2.2 Coloured graph models

2.2.1 CODASYL scheme

As a next step forward in complexity we can take directed graphs with coloured adges. Consider e.g. a simplified CODASYL scheme containing two kinds of set types namely the linked, called "chain", and the pointer array, called "indexed", versions see e.g. fig.2.2 .

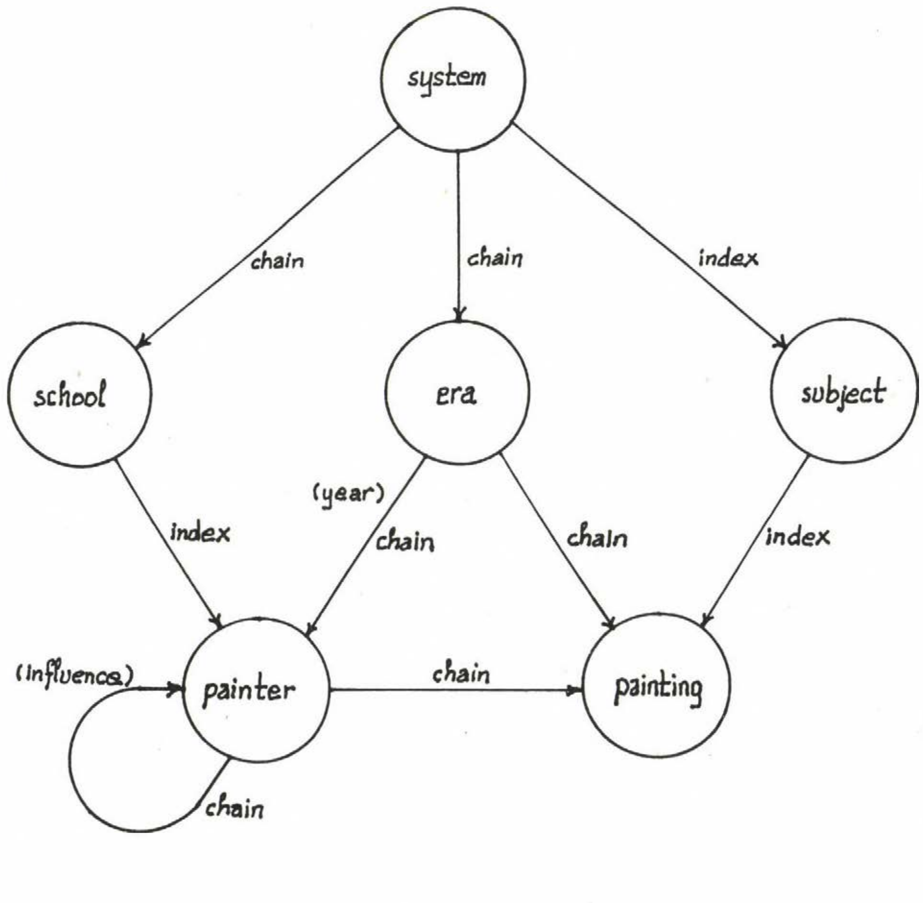


Fig. 2.2

To represent schemes like above we can introduce concepts such as:

```
(2.4)     concept record;

          concept set(owner:record,member:record);

          concept chain is set;
          form member: owned-by owner linked;

          concept array is set;
          form member: owned-by owner indexed;
```

Now the description of the scheme outlined in fig. 2.2 will look like:

```
(2.5)     school: record;
             owned-by system linked;

          era: record;
             owned-by system linked;

          subject: record;
             owned-by system indexed;

          painter: record;
             owned-by school indexed;
             year: owned-by era linked;
             influence: owned-by painter linked;

          painting: record;
             owned-by painter linked;
             owned-by era linked;
             owned-by subject indexed;
```

2.2.2 Chemical factory

Another version may be the colouring of vertices what we illustrate by chemical processes where vertices can represent equipments while edges can streams of materials. Equipments can, in this case, be classified in the following way shown in fig. 2.3 .

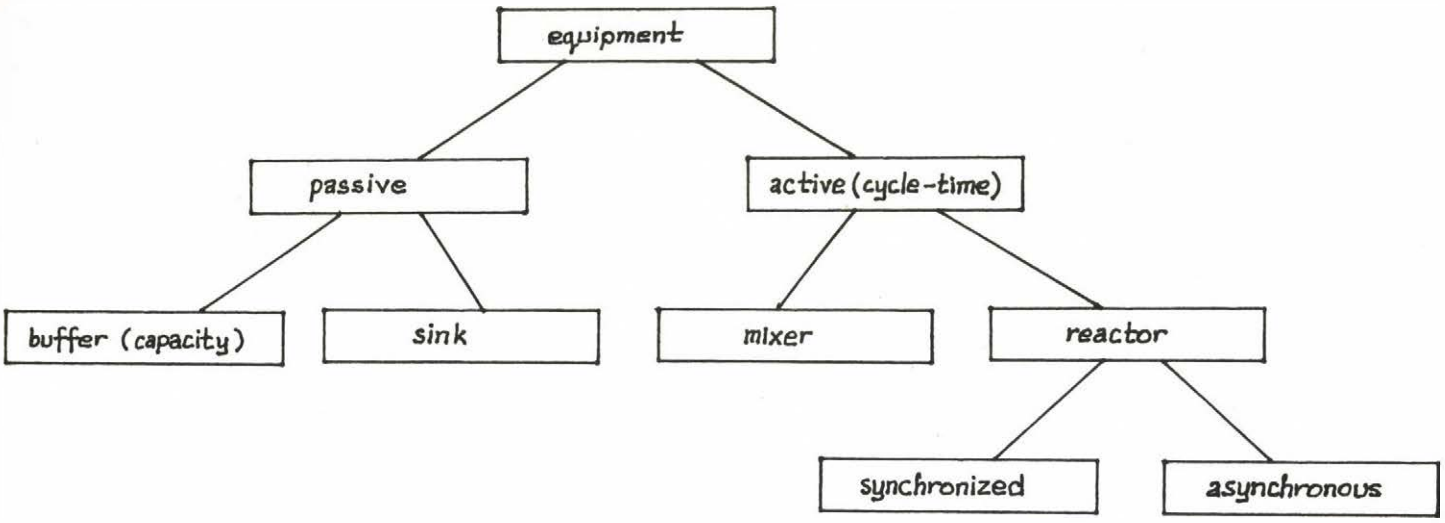


Fig. 2.3

The hierarchy shown can be described by the following set of concepts:

```
(2.6)     concept equipment;

          concept passive-equipment is equipment;

          concept buffer is passive-equipment(capacity:integer);
          form absolute: buffer size capacity;

          concept sink is passive-equipment;
          form absolute: sink;

          concept active-equipment is equipment(cycle:integer);

          concept mixer is active-equipment;
          form absolute: mixer period cycle;

          concept reactor is active-equipment;

          concept synchronized-reactor is reactor;
          form absolute:reactor synchronized by cycle cycle;

          concept asynchronous-reactor is reactor;
          form absolute: asynch reactor work cycle cycle;

          concept stream;
concept output(stream,from:equipment);
form from: output stream stream;

concept input(stream,to:equipment);
form to: input stream stream;
```

Now a chemical process represented e.g. in fig. 2.4

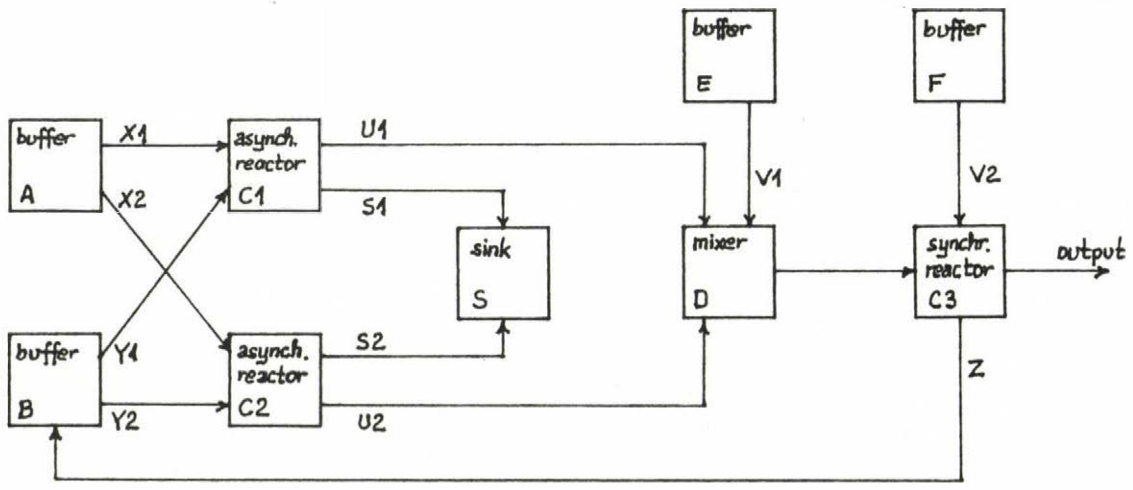


Fig. 2.4

can be described as

```
(2.7)      A: buffer size 30;
              output stream X1,X2;

           B: buffer size 20;
              output stream Y1,Y2;

           C1: asynch reactor work cycle 10;
               input stream X1,Y1;
               output stream U1,S1;

           C2: asynch reactor work cycle 10;
               input stream X2,Y2;
               output stream U2,S2;

           S: sink;

           E: buffer size 10;
              output stream V1;

           F: buffer size 10;
              output stream V2;
```

```
D: mixer period 30;
   input stream U1,V1,U2;
   output stream W;

C3: reactor synchronized by cycle 30;
   input stream W,V2;
   output stream output,Z;
```

2.3 Bipartite graphs

This group of conceptual models is fairly wide, it contains all the so called "net models" e.g. causal nets, place-transition nets, condition-event nets, predicate-transition nets, Petri nets, etc.

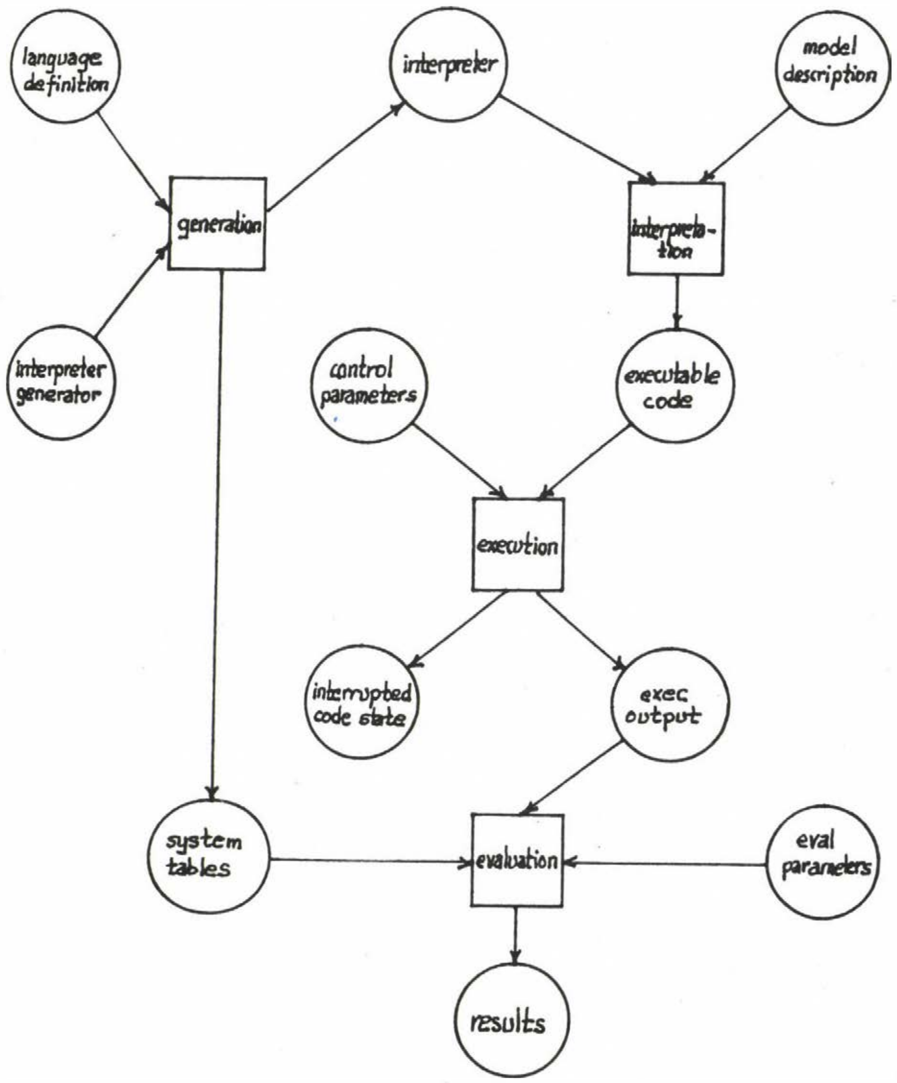


Fig. 2.5

An example for the simplest one of them (a so called causal net) is shown in fig.2.5 expressing a rather typical scheme of meta driven modelling. The characteristic property of such nets is that they do not contain so called "conflicts" i.e. their "condition--edges" (denoted by circles) have at most one input and at most one output arrows.

To represent causal nets we need the following concepts:

```
(2.8)      concept process;
           form absolute: process;

           concept condition;

           concept use(process,condition);
           form process: uses condition;
           form condition: used-by process;

           concept generation(process,condition);
           form process: generates condition;
           form condition: generated-by process;
```

Based on these we can write

```
(2.9)      generation: process;
              generates interpreter;
              uses language-definition, interpreter-generator;

           interpretation: process;
              generates executable-code;
              uses interpreter, model-description;

           execution: process;
              generates interrupted-code, exec-output;
              uses executable-code, control-parameters;

           evaluation: process;
              generates results;
              uses system-tables, exec-output, eval-parameters;
```

Another more sophisticated net models can be represented using pretty similar techniques. Therefore, we disregard of further examples at this point.

2.4 The SADT model

The well known SADT model can be considered to be one of the most advanced conceptual models. As a bipartite directed graph model it has a very important feature, namely the hierarchic decomposability of both its fundamental elements (action and data). Another special (but less important) feature is the classification principle for connections (input, control, etc.). The skeleton of a proposed version of the SADT concepts can be the following:

```
(2.10)      concept diagram;

            concept actigram is diagram
            form absolute: actigram;

            concept datagram is diagram;
            form absolute: datagram;

            concept process(in:actigram,part-of:process);
            default in inherited-from part-of;
            form in: process;
            form part-of: subprocess;

            concept pa-belong(proc:process,into:actigram);
            function;
            implies proc(in=into);

            concept p-sub(proc:process,of:process);
            function;
            implies proc(part-of=of);
            form of: contains proc;


            concept data(in:datagram,part-of:data);
            default in inherited-from part-of;
            form in: data;
            form part-of: subdata;

            concept dd-belong(data,into:datagram);
            function;
            implies data(in=into);

            concept d-sub(data,of:data);
            function;
            form of: contains data;

            concept connection(process,data);

            concept use is connection;
            form process: uses data;
            form data: used-by process;

            concept control is use;
            function;
            form process: controlled-by data;
            form data: controlls process;
```

```
          concept derivation is connection;
          function;
          form process: derives data;
          form data: derived-by process;
```

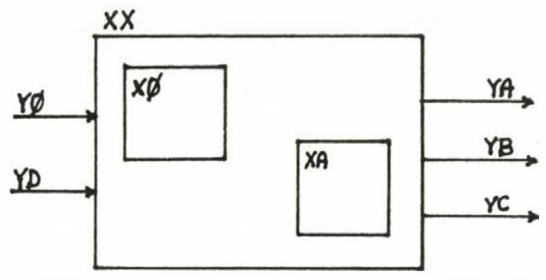
The following example illustrates the use of these concepts.

```
(2.11)    TT-AØ: actigram;
             XX: process;
                uses YØ,YD;
                derives YA,YB,YC;
                contains XØ,XA;

          TT-A1: actigram;
             XØ: process;
                uses YØ;
                derives YA,YB,YE;
                contains XØ1,XØ2;
             XA: process;
                uses YE,YD;
                controlled-by YB;
                derives YC;
                contains XAA;

          TT-D1: datagram;
             YA: data;
                derived-by XØ;
                used-by X1;
                contains YB,YE;
             YD: data;
                derived-by XB;
                used by XA,XAA;
```

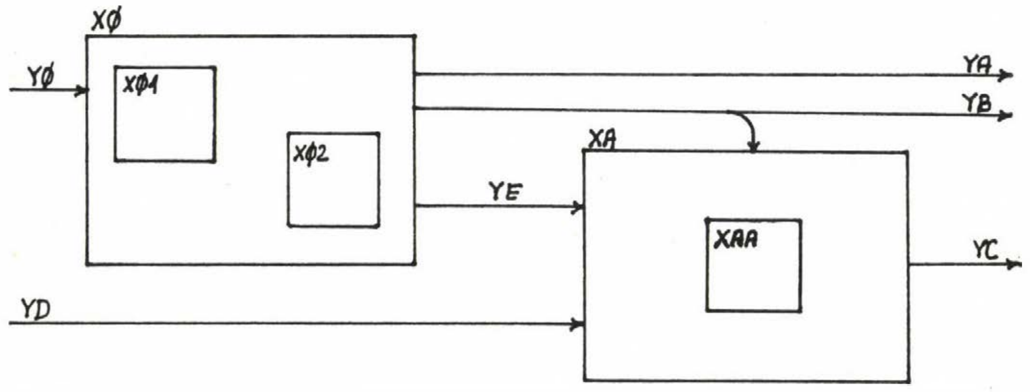
Diagrams represented above may look like as fig. 2.6 .

Note that semantical information expressed by a model is usually highly depend on the selection of concepts and their constraints associated. This holds for the SADT model either. Some basic semantic connections are captured in (2.10) but there are much further possibilities in the model. We shall not detail this problem here, however.

TT-AØ:



TT-A1:



TT-D1:

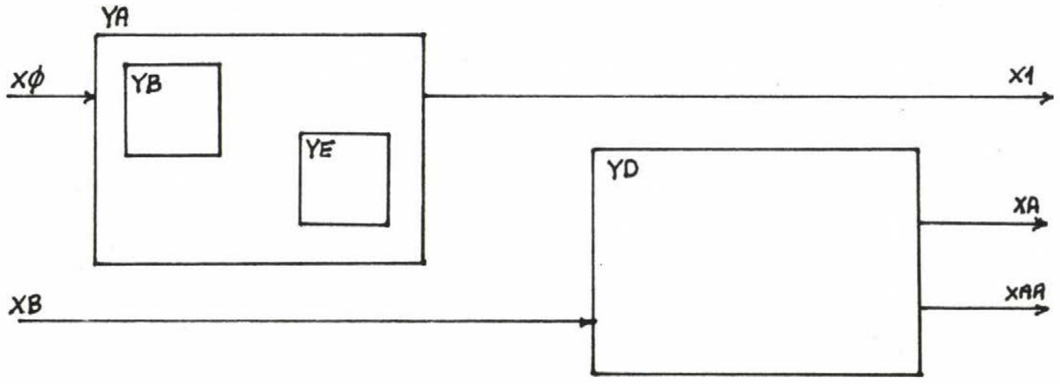


Fig. 2.6

3. ADVANCED LOGICAL MODELS

In the so called logical level of modelling we still have the underlying basic graph-structures but greit variety of further concepts and their refinements is introduced in order to reflect system structures in all logically important details (but still neglecting irrelevant implementation details).

All these models are necessarily problem oriented for the rich choice of concepts does not make it possible to apply a kind of "overall general" approach.

3.1 Information system design

The most advanced model for information systems logical design is the ISDOS PSL/PSA system. By the help of its main ideas here we only outline the roots of a possible approach for information system descriptions.

3.1.1 Basic concepts

Fig.3.1 shows a version of the concepts' subordination scheme which can be used to reflect some most fundamental aspects of information systems.

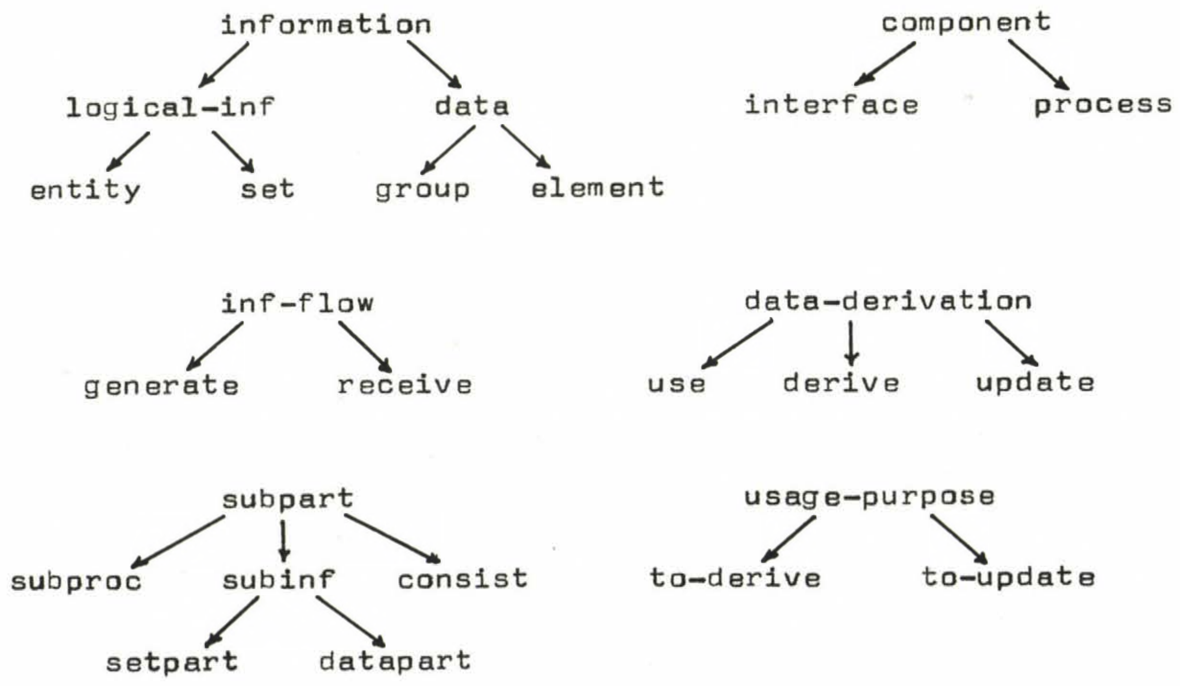


Fig.3.1

Among the main elements shown in fig.3.1 the attribute connections can be defined in the following way

```
(3.1)     inf-flow(component,information)
          data-derivation(process,information)
          usage-purpose(used:infromation,process,towards:information)
          to-derive  ⇒ derivation(process,towards)
                     ⇒ use(process,used)
          to-update  ⇒ update(process,towards)
                     ⇒ use(process,used)
          setpart(part:logical-inf,of:set);
          datapart(part:data,of:group);
          consist(logical-inf,of:data);
```

Having accepted these main lines our decriptive language can be defined in the following way:

```
(3.2)     concept information;
          form absolute: information;

          concept logical-inf is information;

          concept entity is logical inf;
          form absolute: entity;

          concept set is logical-inf;
          form absolute: set;

          concept data is information;

          concept element is data;
          form absolute: element;

          concept group is data;
          form absolute: group;



          concept component;

          concept interface is component;
          form absolute: interface;

          concept process is component;
          form absolute: process;
```

```
concept inf-flow(component,information); function;

concept generate is inf-flow;
form component: generates information;
form information: generated-by component;

concept receive is inf-flow;
form component: receives information;
form information: received-by component;


concept data-derivation(process,information); function;

concept use is data-derivation;
form process: uses information;
form information: used-by process;

concept derive is data-derivation;
form process: derives information;
form information: derived-by process;

concept update is data-derivation;
form process: updates information;
form information: updated-by: process;


concept subpart;

concept subproc is subpart(process,of:process);
function;
form process: part-of of;
form of: subpart process;

concept subinf is subpart;

concept setpart is subinf(part:logical-inf,of:set);
function;
form part: part-of of;
form of: subpart part;

concept datapart is subinf(part:data,of:group);
function;
form part: part-of of;
form of: subpart part;

concept consist is subpart(logical-inf,of:data);
function;
form logical-inf: consists-of of;
form of: contained-by logical-inf
```

```
concept usage-purpose(used:information,user:process,
                      towards:information);

concept to-derive is usage-purpose; function;
implies derive(user,towards);
implies use(user,used);
form used: used-by user to-derive towards;
form user: uses used to-derive towards;
form towards: derived-by user using used;

concept to-update is usage-purpose; function;
implies update(user,towards);
implies use(user,used);
form used: used-by user to-update towards;
form user: uses used to-update towards;
form towards: updated-by user using used;


concept relation;
form absolute: relation;

concept associated(data,to:relation); function;
form to: associated-data data;
form data: associated-to to;

concept maintain(relation,process); function;
concept process: maintains relation;
concept relation: maintained-by process;

 etc.
```

3.1.2 Payroll-system: example

Now we outline some details of the payroll-system example originated from the ISDOS project.

a) System flow:

```
employee-inf: information;

paysys-outps: information;

payroll-master-inf: set;

departments-and-employees: interface;
   generates employee-inf;
   receives paysys-outps;

payroll-processing: process;
   updates payroll-master-inf;
   receives employee-inf;
   generates paysys-outps;
```

b) System structure:

```
        employee-inf: set;
            subpart entity time-card,
                    entity hourly-employment-form,
                    entity salaried-employment-form,
                    entity employment-termination-form;
                    etc.

        paysys-outps: set;
            subpart error-listing,pay-statement,
                    hourly-employee-report,etc.;

        payroll-processing: process;
            subpart new-employee-processing,
                    hourly-employee-processing,
                    salaried-employee-processing;
                    etc.

        new-employee-processing: process;
            receives hourly-employment-form,
                     salaried-employment-form,
                     tax-withholding-certificate;
            generates hired-report;

            etc.
```

c) Data structure:

```
        time-card: entity;
            consists-of employee-name,
                        security-number,
                        status-code,
                        pay-date;
                        etc.

        personal-data: group;
            subpart employee-name,
                    security-number,
                    sex,
                    birthdate;
                    etc.

            etc.
```

d) Data derivation:

```
        hourly-employee-processing: process;
            uses time-card,h-emp-rec;
            derives pay-statement,error-list,etc.;
            subpart h-paycheck-validation,
                    hourly-emp-update;
                    etc.
         etc.       ...
```

3.1.3 Alternative representations

In this point we briefly discuss two further possibilities to represent the "usage-purpose" structure introduced in (3.2). We can use e.g. the following version too:

$$\begin{tikzcd}
 & \text{use(process,used:information)} & \\
 & \text{usage-purpose(use,towards:information)} \arrow[dl] \arrow[dr] & \\
\text{to-update} & & \text{to-derive}
\end{tikzcd}$$

Fig.3.2

that is

```
(3.3)     concept use(process,used:information); function;
          form process: uses used;
          form used: used-by process;

          concept usage-purpose(use,towards:information);

          concept to-derive is usage-purpose; function;
          form use: to-derive towards;

          concept to-update is usage-purpose; function;
          form use: to-update towards;
```

and resulting in description e.g. like

```
(3.4)     P: process;
             uses data D;
                 to-derive E;
                 to-update F;
```

Another version may be:

$$\begin{tikzcd}
 & \text{use(process,used:information)} \arrow[d] & \\
 & \text{usage-with-purpose(towards:information)} \arrow[dl] \arrow[dr] & \\
\text{use-to-derive} & & \text{use-to-update}
\end{tikzcd}$$

Fig.3.3

i.e. more exactly

```
(3.5)     concept use(process,used:information);

          concept use-with-purpose is use(towards:information);

          concept use-to-derive is use-with-purpose; function;
          implies derivation(process,towards);
          form process: uses used to-derive towards;
          etc.

          concept use-to-update is use-with-purpose; function;
          implies update(process,towards);
          form towards: updated-by process using used;
          etc.
```

which can be used e.g. as

```
(3.6)     P: process;
               uses D to-derive E;

          D: data;
               updated-by P using E;

          etc.
```

3.2 Process control systems

3.2.1 Activity decomposition

In this paragraph we only briefly outline the fundamental activity structure of the process controll spacifioation language PCSL. Fig.3.4 shows the hierarchy introduced.

$$\begin{tikzcd}
 & \text{activity(utilize:activity)} \arrow[dl] \arrow[d] \arrow[dr] & \\
\begin{array}{l}\text{paralelly}\\ \text{decomposed}\\ \text{activity}\end{array} & \begin{array}{l}\text{sequentially}\\ \text{decomposed}\\ \text{activity}\end{array} & \begin{array}{l}\text{alternatively}\\ \text{decomposed}\\ \text{activity}\end{array}
\end{tikzcd}$$

Fig.3.4

There are several ways to introduce sentence forms to make descriptions having the above type-structure. Now we show an "owner-oriented" way (for the main part of it) ensuring hierarchically arranged description formats.

```
(3.7)   concept activity(alternative-of:alternative,
                         step-of:sequence,
                         co-element-of:parallel,
                         seq-no:integer);
        form alternative-of: alternative;
        form step-of: step seq-no activity;
        form co-element-of: coactivity;

        concept alternative is activity;
        form alternative-of: a-alternative;
        form step-of: step seq-no a-activity;
        form co-element-of: a-coactivity;
        form absolute: a-activity;

        concept sequence is activity;
        form alternative-of: s-alternative;
        form step-of: step seq-no s-activity;
        form co-element-of: s-coactivity;
        form absolute: s-activity;

        concept parallel is activity;
        form alternative-of: p-alternative;
        form step-of: step seq-no p-activity;
        form co-element-of: p-coactivity;
        form absolute: p-activity;
```

Using these concepts the following scheme

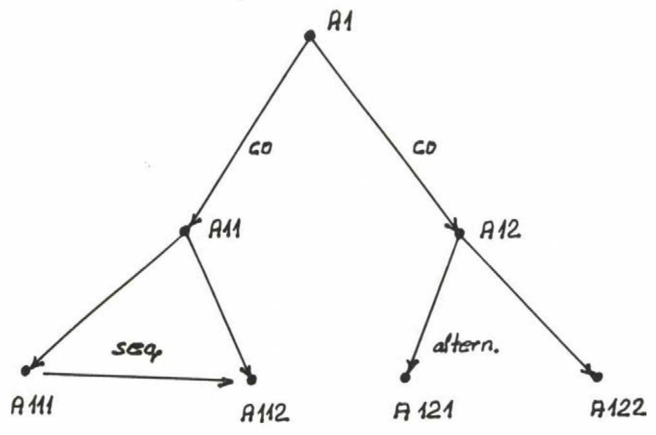


Fig.3.5

can be described as

```
(3.8)     A1: p-activity;
              A11: s-coactivity;
                   A111: step 1 activity;
                   A112: step 2 activity;
              A12: a-coactivity;
                   A121: a-activity;
                   A122: a-activity;
```

3.2.2 Process communication

There are three most important communication desciplines between processes namely the "mutual exclusion" associated with resources, the so called "reader/writer scheme" associated with shared storages, and the so called "producer/consumer scheme" associated with buffers.

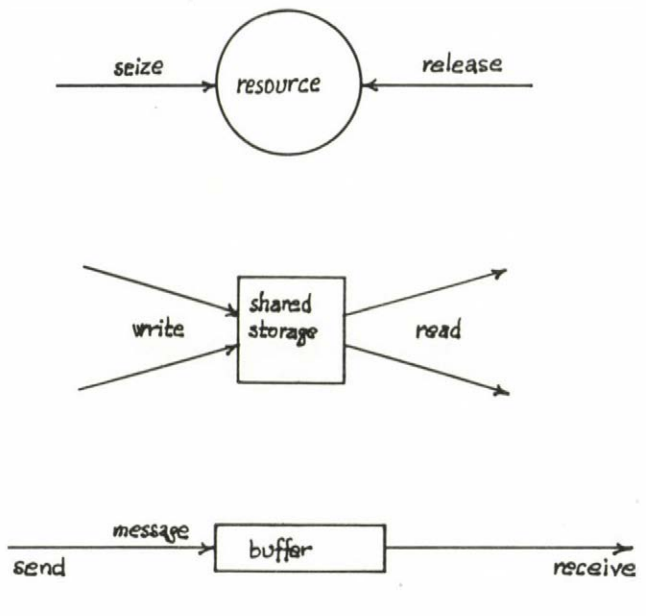


Fig.3.6

The concepts to model these schemes in descriptions may well be the following. (We disregard here of giving form clauses and examples for their use.)

(3.8)

```
concept resource;
concept resource-action(process,resource);
concept seize is resource-action;
concept release is resource-action;


concept shared-data is data;
concept shared-access(process,shared-data);
concept read-access is shared-access;
concept write-access is shared-access;


concept buffer;
concept message;
concept buffer-action(process,buffer,message);
concept produce is buffer-action;
concept consume is buffer-action;
```

4. MACHINE LEVEL DESCRIPTIONS

At this level we deal with machine level structural details of systems to be implemented. We must emphasize however, that while at the conceptual and logical levels description in itself has of fundamental importance, it is not so obvious at the machine level. Here it is always worth considering whether a more machine close tool providing full semantical analysis (e.g. the programming language itself, or a VDM-like semantic specification language) is preferable instead.

Therefore, in general, making machine level descriptions is not the recommended application of our description tools. It can be, however, when our intention is to use it just for:

a) the analysis of structural logic, and/or

b) documentation purposes.

4.1 Control structures

From the viewpoints of structural analysis we are not at all interested in particular machine instructions, but in program control primitives only. We shall consider a version of structured programming using some fairly common notations.

4.1.1 Sequence

To start with, blocks of sequences are represented using the pattern shown in fig. 4.1 .

$$\begin{tikzcd}
\text{statement(subord-to:control-statement)} \arrow[d] \\
\text{control-statement} \arrow[d] \\
\text{sequence}
\end{tikzcd}$$

Fig. 4.1

where there is only one type of "control-statement" introduced namely that called "sequence", and we do not deal with other statement forms and kinds (for the time being) at all. Now supposing the above concepts are declared as

```
(4.1)     concept statement(subord-to:control-statement);
          form subord-to: statement;

          concept control-statement is statement;
          concept sequence is control-statement;
          form absolute: seq;
          form subord-to: seq;
```

we can write nested blocks e.g. as

```
(4.2)     PROG: seq;
             A: statement;
             B: statement;
             C: seq;
                CA: statement;
                CB: statement;
             end;
             D: statement;
          end;
```

4.1.2 Iteration

Let us now enrich fig. 4.1 as

$$\begin{tikzcd}
 & \text{statement(subord-to:control-statement)} \arrow[d] & & \\
 & \text{control-statement} \arrow[dl] \arrow[dr] & & \\
\text{sequence} & & \text{iteration(condition:text)} \arrow[dl] \arrow[dr] & \\
 & \text{while} & & \text{until}
\end{tikzcd}$$

Fig. 4.2

and supplement (4.1) by

```
(4.3)     concept iteration is control-statement(condition:text);

          concept while is iteration;
          form absolute: iter-while condition;
          form subord-to: iter-while condition;

          concept until is iteration;
          form absolute:  iter-until condition;
          form subord-to: iter-until condition;
```

Using (4.1) and (4.3) together we can write structures like:

```
(4.4)       iter-until "S=limit";
               A: statement;
               B: statement;
            end;
```

```
(4.5)       seq;
               A: statement;
               B: iter-while "key(S)≠high values";
                  BA: statement;
                  iter-while "0 < S ≤ NUM-E";
                     BBA: statement;
                  end iter;
               end iter;
            end seq;

            etc.
```

4.1.3 Selection

To get an appropriate way for the representation of selection in the sense e.g. of Jackson we can introduce the following pattern of concepts:

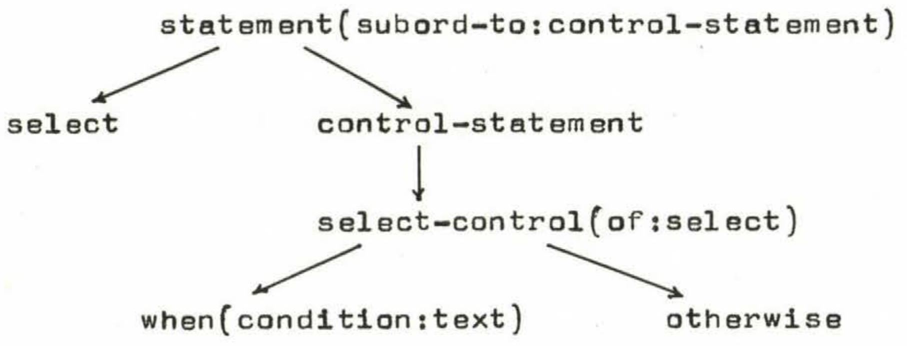

statement(subord-to:control-statement)
select
control-statement
select-control(of:select)
when(condition:text)
otherwise

Fig. 4.3

represented e.g. as

```
(4.6)       concept statement(subord-to:control-statement);
            form subord-to: statement;

            concept select is statement;
            form absolute: select;
            form subord-to: select;

            concept select-control is control-statement(of:select);

            concept control-statement is statement;
concept when is select-control(condition:text);
form of: when condition;

concept otherwise is select-control;
form of: otherwise;
```

and used e.g. as

```
(4.7)      select;
              when "code='C'";
                 CREDIT: statement;
                 COUNT:  statement;
              when "code='T'";
                 TRANSFER: statement;
              otherwise;
                 ERROR-1: statement;
           end select;
           etc.
```

4.1.4 Parallelism

Parallel execution structure can be expressed very similarly:

$$\begin{tikzcd}
 & \text{statement(...)} \arrow[dl] \arrow[dr] & \\
\text{parallel-block} & & \text{control-statement} \arrow[d] \\
 & & \text{parallel-seq(in:parallel-block)}
\end{tikzcd}$$

Fig. 4.4

defined e.g. like:

```
(4.8)      concept statement(subord-to:control-statement);
           form subord-to: statement;

           concept parallel-block is statement;
           form absolute: parblock;
           form subord-to: parblock;

           concept parseq is control-statement(in:parallel-block);
           form in: co;
```

and used e.g. like

```
(4.9)     parblock;
             co;
                A: statement;
                B: statement;
             co;
                C: statement;
                D: statement;
          end parblock;
```

4.1.5 Structured programs: summary

A summary of the structures proposed in 4.1.2 - 4.1.4 based on iteration, selection, and parallel-execution can be the following. (We shall not use the structure "sequence" here explicitely for it can be replaced by just sequential writing which is automatically adequate in the present formalism.)

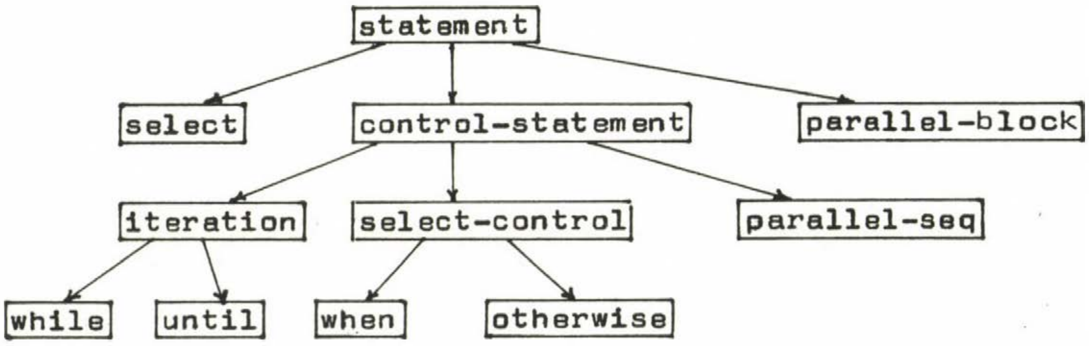


Fig. 4.5

The complete forms of the concepts shown above can be the following:

```
(4.10)    concept statement(in:control-statement);
          form in: statement;

          concept select is statement;
          form absolute: select;
          form in: select;

          concept parallel-block is statement;
          form absolute: parblock;
          form in: parblock;
```

```
concept control-statement is statement;

concept iteration is control-statement(cond:text);

concept while is iteration;
form absolute: iter-while cond;
form in: iter-while cond;

concept until is iteration;
form absolute: iter-until cond;
form in: iter-until cond;

concept select-control(of:select);

concept when is select-control(cond:text);
form of: when cond;

concept otherwise is select-control;
form of: otherwise;

concept parallel-seq is control-statement(of:parallel-
form of: co;                                        block);
```

4.2 Data structures

4.2.1 Syntactic representation

We may have seen that concepts expounded in (4.10) were based on the three so called regular operations, as it is always the case in all versions of structured programs. It is well known, moreover, that syntactic structures (of context free languages) in general can also be described by regular expressions. Therefore, we can describe data syntax by analogous terms as in (4.10). Consider e.g. the syntactic structure expressed in fig. 4.6 using the formalism of jackson.

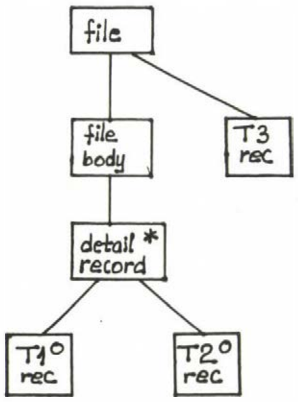


Fig. 4.6

Recalling the meaning of Jackson's formalism fig. 4.6 expresses that

1) "file" is a sequence of two members: "file-body" and "T3";
2) "file-body" consists of (zero or more) repetitions of "detail-records";
3) a detail record is either a "T1" or "T2" record.

To describe structures like these formally we can introduce the following versions of regular expressions for data

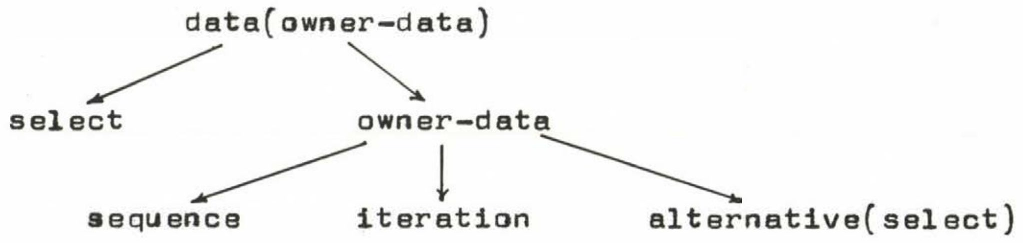


Fig. 4.7

with the following forms

```
(4.11)      concept data(owner-data);
            form owner-data: data;

            concept select is data;
            form absolute: select;
            form owner-data: select;

            concept owner-data is data;

            concept sequence is owner-data;
            form absolute: sequence;
            form owner-data: sequence;

            concept iter is owner-data;
            form absolute: iter;
            form owner-data: iter;

            concept alternative is owner-data(select)
            form select: either;
            form select: or;
```

Now, using (4.11) we can write fig. 4.6 as

```
(4.12)     file: sequence;
              file-body: iter;
                 detail-record: select;
                    either;
                       T1: data;
                    or;
                       T2: data;
                 end select;
              end iter;
              T3: data;
           end sequence;
```

4.2.2 Structural representation

In this point we outline a way of describing physical data in accordance with the conventions of COBOL and of the ANSI X3 DDL. The skeleton we give here can provide a basis for designing either conventional file systems or CODASYL databases.

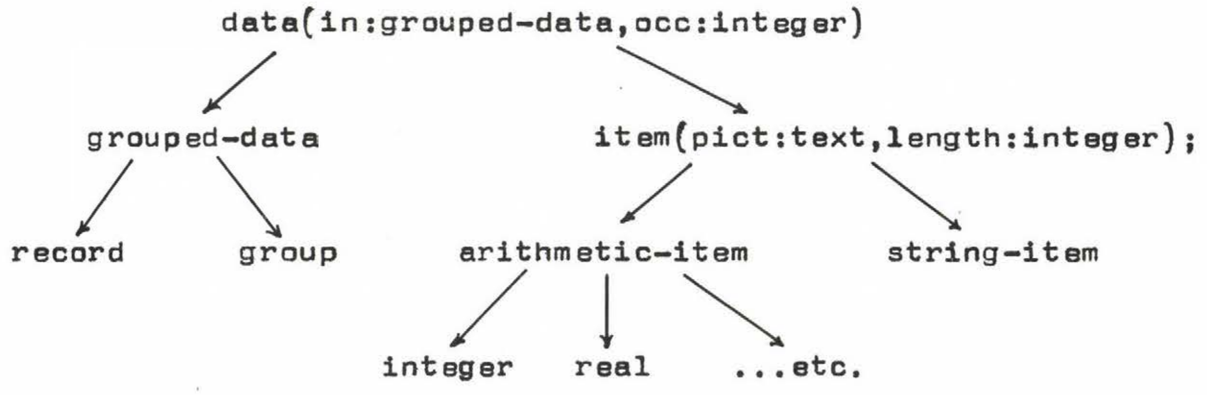


Fig. 4.8

that is, more exactly:

```
(4.13)     concept data(in:grouped-data,occures:integer);

           concept grouped-data is data

           concept record is grouped-data;
           form absolute: record;

           concept group is grouped-data
           form in: group;
           form in: group occ occures;
```

```
concept item is data(picture:text,length:integer);

concept string-item is item;
form in: text length;
form in: text length pict picture;
form in: text length occ occures;
form in: text length pict picture occ occures;

concept arithmetic-item is item;

concept real is arithmetic-item;
form in: real length;
form in: real length pict picture;
form in: real length occ occures;
form in: real length pict picture occ occures;

concept integer is arithmetic-item;
form in: integer length;
form in: integer length pict picture;
form in: integer length occ occures;
form in: integer length pict picture occ occures;

concept key(of:record,is:item);
function;
form of: key is;
```

As a brief illustration for the use of the concepts introduced above we can consider:

```
(4.14)     R1: record;
               key D2;

               D1: real 4 pict ZD.D occ 12;
               D2: integer 3;
               D3: group occ 5;
                   D31: integer 2 occ 7;
                   D32: text 3 pict XXX;
           end record;
```

4.2.3 File systems

When thinking in conventional file systems the most important thing is the description of file definitions. Fig.4.9 outlines one possible root for a file-type hierarchy.

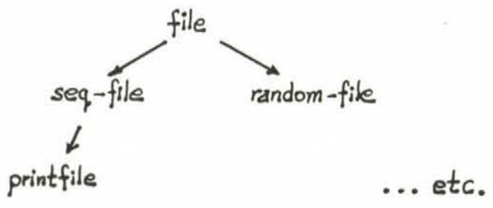


Fig. 4.9

Using fig.4.9 we can supplement (4.13) by the following definitions

```
(4.15)      concept file(blksize:integer);

            concept random-file is file;
            form absolute: random-file;

            concept sequential-file is file;
            form absolute: seq-file;

            concept printfile is sequential-file(linelength:integer);
            form absolute: printfile;

            concept record(of:file);
            form of: record;

            concept program;

            concept access(program,file,mode:text);
            function of (program,file);
            form file: accessible from program mode mode;
```

To illustrate the use of the concepts introduced we can write e.g.

```
(4.16)      orders: seq-file;
               order: record;
               accessible from daily-list mode "input";
               accessible from consume-o1 mode "output";

            enterprises: random-file;
               firm: record;
               accessible from consume-o1 mode "hashed-input";
            etc.
```

4.2.4 Database management systems

Concepts outlined in 4.2.2 for data structures can also be used to make dascriptions for set oriented databases. Some of the main concepts we need may be the following:

```
(4.17)      concept set(owner:record,mode:text);
            form absolute: set;

            concept membership(record,set);
            function;
            form record: member-of set;
            form set: member record;

            concept key(membership,item);
            form membership: key item;
```

```
concept search-key is key;
form memberchip: search-key item
```

Of course one can introduce various set types too, and other necessary set attributes to meet exactly one of the proposed standards.

Because set oriented databases are so well known today, we do not think it is necessary to illustrate the use of the above concepts.

APPENDIX: FUNDAMENTAL TECHNIQUES

Referring to the description methodology provided by the system SDLA we give here some techniques generally applicable in various fields and models.

A.1 Clause blocking

Suppose we are to write series of clauses (disregarding of their kinds and meaning at this level of abstraction), and wish to group them into blocks in arbitrary depth of nesting. This can simply be achieved by the pattern of concepts:

$$\begin{tikzcd}
\text{clause(subord-to:block)} \arrow[d] \\
\text{block}
\end{tikzcd}$$

Fig.A.1

to be declared as

```
(A.1)      concept clause(subord-to:block);
           form subord-to: clause;

           concept block is clause;
           form absolute: block;
           form subord-to: block;
```

These forms allowe writing structures such as

```
(A.2)      block;
              A: clause;
              B: clause;
              C: block;
                   CA: clause;
                   CB: clause;
              end;
              D: clause;
           end;

           etc.
```

A.2 Recursive structures

The following simple pattern can be used wherever recursivity is necessary (either for data, or processes or whatever).

origin(element,father)

$$\begin{tikzcd}
 & \text{element} \arrow[dl] \arrow[dr] & \\
\text{terminal} & & \text{father}
\end{tikzcd}$$

Fig.A.2

that is

```
(A.3)       concept origin(element,father); function;
            concept element;
            concept father is element;
            concept terminal is element;
```

Now every element can have a father which is also an element, but terminal elements may not have sons.

A.3 Attributes versus relations

Sometimes we can not decide whether to use an attribute or a self-contained concept in order to represent a given connection, because in some contexts one of them is better while in others the other one. This can be resolved by defining both and using the constraint shown below:

```
(A.4)       concept thing(owner:thing);

            concept ownership(member:thing,of:thing);
            implies member(owner=of);
```

(Note, however, that we do not introduced the inverse constraint above and neither the analysis facilities need it.)

A.4 Exclusion by form clauses

The declaration of forms for concepts may provide additionally for a semantic check function by excluding undesirable subordinations by the lack of forms in certain contexts.

We refer to (4.13) and fig.4.8 where "record" conceptually may be embedded into another "record" for it is a subtype of "data", however, the lack of the corresponding relative form excludes such an illegal usage at the user level.

Literature

1. Alagic, S., Arbib, M.A.: The Design of Well-Structured and Correct Programs. Springer-Verlag. 1977.

2. ANSI X3H2 - Data Description Language. Jan. 198o.

3. Balogh, K., Farkas, Zs., Sántáné - Tóth, E., Szeredi, P.: LDM tervező rendszer, SOFTTECH, 1979.

4. Bauer, F.L. /ed/: Software Engineering. Springer, 1975.

5. Baumann, R.: Computer-aided design and implementation of control algorithms. IFAC 78, I. Helsinki, Jun.1978.

6. Bernus, P.,: Representation of SADT in SDLA. Private communication.

7. Bernus, P.: Connecting SADT and ISDOS into a system design system. MTA SZTAKI, Budapest, Jan. 1979.

8. Biewald, J.P., et al.: EPOS- A specifications and design technique for computer controlled real-time automation systems. Proc. 4 th Int.Conf. Sofw. Engineering, Munich, Sept. 1979.

9. Bjorner, D., Jones, C.B. /ed/: The Vienna Development Method: The Meta Language. Lecture Notes in Computer Science, 61, Springer, 1978.

1o. Codd, E.F.: Extending the database relational model to capture more meaning

11. Constantine, L.L., et al.: Structured design, IBM Sys.J. 13,2, 1974.

12. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, 1972.

13. Dahl, O.J., Nygard, K., Myrhaugh, N.: SIMULA67 Common Base. NCC, Oslo, 197o.

14. David, K., Jefferson, P.D.: Analysis of requirements for a large-scale information system. Proc. COMPSAC'79, Chicago, 1979.

15. Dijkstra, E.W., Wirth, N.: Stepwise refinement. CACM 14, 4, 1971.

16. Eszto,Z., et al: Számitógépes információrendszerek tervezési és módszertani eszközei. KSH SZÁMOK, Budapest, 1975.

17. Gries, D.: Programming by induction. Information Processing Letters, 1,3, 1972.

18. Halassy, B.: SZIAM általános leirás. KSH SZÁMOK, Budapest, 198o.

19. Jackson, M.A.: Principles of Program Design. Academic Press, 1975.

2o. Knuth, E., Radó, P., Tóth, Á.: Preliminary description of SDLA. MTA SZTAKI, Tanulmányok 1o5, Budapest, 198o.

21. Knuth, E., Rónyai, L.: Closed convex reference schemes. MTA SZTAKI, WP-II/12, Nov, 198o.

22. Knuth, E., Radó, P., Rónyai, L.: Relational Query language in reference environment, MTA SZTAKI, WP-II/9, March, 198o.

23. Ludewig, J., Streng, W.: Methods and tools for software specification and design - A survey. Eur. Purdue Workshop, TC7, Zürich, April 1978.

24. Ludewig, J.: PCSL - A process control software specification language. Kernforschungszentrum Karlsruhe GmbH. April 198o.

25. Mérey, A. /szerk/: A MOZ-ART programozási technológia. KSH SZÁMOK-SZÁMKI, SOFTTECH D44, Dec. 1979.

26. Myers, G.J.: Software Reliability. Wiley Sens, 1976.

27. Naur, P.: Programming by action clusters, BIT 8,3 1969.

28. Neuhold, E.J. /ed/: Formal Description of Programming Concepts. North-Holland, 1977.

29 Nora, S., Minc, A.: L'informatisation de la société. Rapport á, M.le Président de la République. Paris, 1978.

3o. Parnas, D.L.: A technique for the specification of software modules with examplex. CACM 15, 5, 1972.

31. Shaw, B. /ed/: Computing System Design. University of Newcastle upon Tyne, Sept. 1976.

32. Sundgren, B.: Theory of Data Bases. Petrocelli, New York, 1975.

33. Sztanó, T., Almássy, G., Huppert, A.: Design technique for real-time systems. Private communication.

34. Teichroew, D., E.A. Hershey III: PSL/PSA a computer-aided technique for structured documentation and analysis of information processing systems. IEEE Trans. SE-3, 1,1977.

35. Teichroew, D., Winters, E.: Recent Developments in System Analysis and Design, 1979.

36. Teichroew, D., Knuth, E., Radó, P., Szokolov , M.: Operations within the hierarchic concepts'model. MTA SZTAKI, WP-II/11, Sept, 198o.

37. Tóth, A.: Computer aided design of a maintanence project. Kosice. Private communication.

38. Users manual for Generalized Analyzer B1.2 under MTS. Preliminary draft, ISDOS Project, ref 80G12-0284-2, Jan, 198o.

39. Warnier, J.D.: Lois de construction de programmes. Entrainement a la programmation, Les editions d' organisation, 1971.

<u>List of papers available on project SDLA</u>
/dec. 198o./

1. Knuth, E., Radó, P., Tóth, Á.: Az SDLA előzetes ismertetése. MTA SZTAKI Tanulmányok 1o4, 1979.

2. Knuth, E., Radó, P., Tóth, Á.: Preliminary description of SDLA.
MTA SZTAKI Tanulmányok 1o5, 1979.

3. Radó, P.: Az ISDOS DBMS új szervezése. MTA SZTAKI Working Papers II/2, 1979. szept.

4. Knuth, E., Rónyai, L.: Closed reference schemes. MTA SZTAKI Working Papers II/7, jun. 1979.

5. Knuth, E., Radó, P., Rónyai, L.: Relational query language in reference environment. MTA SZTAKI Working Papers II/9, march, 198o.

6. Radó, P., Kiss, O., Szilléry, A.: Relációs adatbázis interface. MTA SZTAKI Working Papers II/1o, may. 198o.

7. Teichroew, D., Knuth, E., Radó, P., Szokolov, M.: Operations within the hierarchic concepts' model. MTA SZTAKI Working Papers II/11, sept. 198o.

8. Knuth, E., Rónyai, L.: Closed convex reference schemes. MTA SZTAKI Working Papers II/12, oct. 198o.

9. Knuth, E., Radó, P.: Principles of computer aided system description. MTA SZTAKI Working Papers II/13, nov. 198o.

1o. Radó, P., Kiss, O., Knuth, E., Szilléry, A.: SDLA 1.0 users manual. MTA SZTAKI Working Papers II/14, nov. 198o.

A TANULMÁNYSOROZATBAN 1980-BAN JELENTEK MEG:

101/1980	Gerencsér László - Hangos Katalin: Diszkrét lineáris sztochasztikus rendszerek önhangoló szabályozása
102/1980	Pásztorné Varga Katalin: Rekurziv eljárás
103/1980	Gerencsér Piroska - Szép Endre - Zilahy Ferenc Marton Zsolt: Robotmegfogók adaptivitása I.
104/1980	Knuth Előd - Radó Péter - Tóth Árpád: A SDLA előzetes ismertetése
105/1980	E. Knuth - P. Radó - A. Toth: Preliminary description of SDLA
106/1980	Prékopa András: Sztochasztikus programozási modellek és alkalmazásuk
107/1980	Kelle Péter: Megbizhatósági készletmodellek és alkalmazásuk
108/1980	Almásy Gedeon: Mérlegegyenletek és mérési hibák
109/1980	Békéssy A. - Demetrovics J. - Gyepesi Gy.: Relácios adatbázis logikai szintü vizsgálata funkcionális függőségek szempontjából
110/1980	Gaál A. - Soltész J. - Ruda M. - Ratkó I.: Tanulmányok a statisztikai adatfeldolgozásról
111/1980	Benedikt Szvetlána: Nem ismételhető döntéshozatal analizise kockázattal járó esetekben
112/1980	Verebély Pál: Többprocesszoros, osztott intelligenciáju grafikus rendszerek tervezési és megvalósitási kérdései
113/1980	V. Visegrádi Téli Iskola

114/1980	Demetrovics János: Relációs adatmodell logikai és strukturális vizsgálata
115/1980	Gergely József: Program package for sparse matrices

1981-BEN JELENTEK MEG:

116/1980	Siegler András: Egy 6 szabadságfoku antropomorf manipulátor kinematikája és számitógépes vezérlése

MAHIR - ROZMARING MGTSZ. - Nyomda 81.218