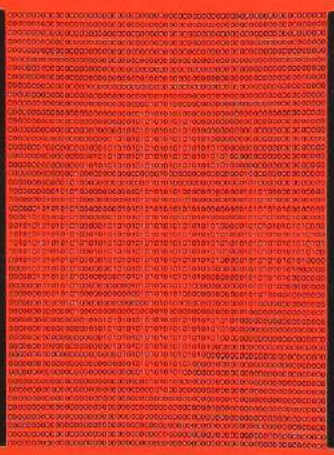
Heft 12/Dezember 1981 S 20756 E



Angewandte Informatik
applied informatics

Aus dem Inhalt

Printed in Germany

V

Vieweg

S & DL – Structured Design Language (Proposal)

Zygmunt Ryznar, Cracow (Polen)

Stichworte: *Spezifikationssprache, strukturierter Entwurf, ergodische Systeme, Einheiten, Strukturierungsmechanismen*

Zusammenfassung: *Es wird eine Spezifikationssprache namens S & DL für den strukturierten Systementwurf vorgeschlagen. Es wird die Syntax dieser Sprache erörtert, und einige Nutzungsbeispiele werden vorgestellt. Es werden Beziehungen, die zum Aufbau komplexer Systeme aus einfachen Bestandteilen erforderlich sind, definiert.*

Key-words: *specification language, structured design, ergodic systems, entity, mechanisms of structuring*

Abstract: *A specification language named S & DL dedicated for the system structured design is proposed. The syntax of this language is discussed and some patterns of usage are presented. Relationships necessary to create complex systems from simple components are defined.*

> *"Commercial DP is under fire. Users are becoming increasingly dissatisfied with the service which their DP departments are producing. Manpower costs are rising, but productivity is often static or actually falling: quality has not improved dramatically..."*
>
> *J. Rhodes*

Introductory remarks

One can risk the statement that the structured design is getting less successful than it was expected. There have been formulated good aims, e.g. "making coding, debugging, and modification easier, faster, and less expensive by reducing complexity" [1], but tools which lead to wide-spread implementation are practically non-existent. The reason is that the structured design prefers techniques to methods, and these techniques have not been correlated with each other. Techniques are centered mostly on some mechanisms of programming (like iteration) or data structures or transformations or output or transactions. The theoretical basis for the structured design is not only a top-down approach, but are also a bottom-up approach, events theory and contingency theory.

If it is assumed that an information system is dedicated to reflect the object reality, it implies some flexibility that allows both programs and files to change in accordance with actual needs of users. To meet this requirement, such systems should behave as an ergodic system, i.e. a system the evolution of which does not depend on its initial state. The most desired form of a functioning computerized information system is evolution by generation of many incarnations (shadows of primary form) which cover additional demands. The crucial point of this solution is how to keep the logically floating structure of the system providing only some necessary transfers between components and seeing all raw resources (data and procedures) as common for many incarnations called problem packages.

Making a change and discovering all effects of it is not an easy task. In conventional solutions such an activity is very labour-consuming even in simple cases. A better way is to have some piece of software called "problem operating system" dealing with the description and modifications all related components. Each component has to fit some standards which make possible analysis and modification. Without these standards and specific technology of component tuning, we have got concepts only and practically nothing to be implemented, understood for designers and helpful for users.

Making problem oriented packages may succeed in time (user's decision cannot wait) and in quality, if it is supported by formal notation and software that helps to describe problems and systems, find components useful for a given application, tune them upon parameters and at last assemble them to the form adequate to the user requirements, languages compilers, DBMS facilities, and computer operating system demands. The vital part of the computer-aided design should be a specification language easily to use to designers and not too complicated for users. The purpose of that language is decreasing the technological gap between changeable needs of users and rigid computer technology.

0013-5704/81/12 0526–08 $ 02.00/0

The S & DL language is a special-purpose language dedicated to describe any component of the system (or problem) and its relations. Structuring the *problem*, as a cognitive activity, may based on top-down approach be giving smaller parts easily to understand. For complementary analysis we recommend to use the nonhierarchic FACT (Functional Analysis Cross Table) method described in [3] which is useful particularly in manufacturing companies and provides overall recognition of the "territory" of problem in terms of resources and functions. Problem description is orientated on reflection of dynamics, expressed in terms of events, actions, and processes.

Structuring the *system* differs from a conventional approach centered on subsystems, functional units, modules, etc. Conventions corresponding to the systems approach usually consider a system as a combination of interrelated (but rather static) parts forming the whole, i.e. having the same or common objectives. In this paper we prefer a situational approach: we do not define system and we are describing the problem packages (instead of systems) created for actual business situations. Development of computerized information system is carried out by successive design and implementation of particular packages that must be relevant to the management requirements and decision making process. The basis for such a structured design is some initial state of system, called pre-system (or "library of possibilities") consisting of fundamental technological elements dedicated to serve typical data structures and operations and built according to rules of structuring.

"Problem operating system" by means of D & L language deals with modification of elements and links them in a variety of combinations needed by a problem package. Structuring programs and data resources is not limited to hierarchic relations that facilitate the creation of more flexible structures. This solution leads us to a homologous system, i.e. a system "developed with any control relationship that does not define a hierarchy of control responsibility, i.e. non-hierarchical system" [2].

S & DL syntax description

S & DL (*S*tructured *D*esign *L*anguage) consists of two subsets: the specification language SL and the design language & DL. SL is a nonalgorithmic language dedicated to describe a variety of objects (called entities) and relations between them. The & DL language is a highly nonprocedural language dedicated to retrieve descriptive information, tune components of pre-system, execute structuring, create interfaces to DBMS and computer operating system. The syntax of & DL has not been developed yet. It may be partially derived from special-purpose languages like Metacobol, DML of CODASYL, Q-U of CDC, etc. A necessary condition for implementing & DL is a rather sophisticated DBMS capable to maintain overlapping and floating relations, store textual descriptions and procedures as well as formatted data, process (tune) text of procedures, retrieve all resources listed above by highly nonprocedural commands and display answer in readable form, etc. & DL language should provide ab extended macro-facility, called Procedure Tuner, for generating versions (incarnations) of generalized modules the description of which contains modifiers belonging to secondary entities. Restrictions and rules of transformation are stored in a specification of entity MODIFIER (name). An incarnation is set up by the command CREATE INCARNATION (name FROM (entity-name) USING (list of modifiers' names and sequence of commands MODIFY concerned with modification of each modifier.

The main subject of this paper is the SL specification language. The SL language is applicable on three levels. Firstly, it can define itself on metadefinition level (see table 1) setting up patterns for the language analyzer. Secondly, it is useful on predefinition level to determine the structure and obligatory phrases within description of entity type. This facility simplifies very much specification for each named entity on the last (third) level of definition. The basic syntactic unit of SL is a definition block which contains sentences and clauses. A scope of the block is determined by DEF and ENDEF delimiters. In a statement (sentence or clause) there are distinguished the following lexical atoms: key-words (e.g. entity types), entity names, attributes, commentaries, delimiters, and special signs.

A subject of the specification is an entity which represents a simple or complex object that can be described and accessed as a relatively independent whole. In terms of the definition block, an entity that appears directly after DEF ⟨entity-type⟩ is called a primary entity and entities which act as attributes are called secondary entities. The entity can be secondary in many definition blocks while a simple block can name it primary. The main description of an entity is associated with the primary entity, but the usage of it can be specified in any block. The task for the SL processor (or DBMS) is the creation of cross-references and concentration of descriptive information written in many places of the specification. A secondary entity that is not defined in its own definition block can operate as attribute only and descriptions of it are not gathered by the SL processor.

An obligatory description of an entity is entered from the primary entity definition block DEF ⟨entity-type⟩ ⟨entity-name⟩. Distributed information about entity appears in so-called "autonomous blocks" having format DEF ⟨block-name⟩ and containing complementary specifications of many entities. Another distributed information is located in primary entity blocks on the level of secondary entities. Owing to this opportunity, entities can be described successively and some degree of completeness is needed only if a design command of the & DL language has to be executed.

The description of entity comprises entity identifiers, attributes, list of secondary entities, clauses of structuring, and comments. It is possible to individualize the description even for one occurrence of an entity by the introduction of an additional identifier called IDKEY

Table 1 Metadefinition of the SL language

N°	syntactic presentation of statement
1	⟨batch⟩ : := ⟨header⟩ ⟨batch-name⟩ ⟨statements⟩ ⟨trailer⟩
1.1	⟨SL-batch⟩ : := BEGIN ⟨batch-name⟩ ⟨definition-blocks⟩ END
2	⟨definition-block [def-block⟩ : ⟨primary-entity-def-block⟩/& ⟨autonomous-def-block⟩
3	⟨primary-entity-def-block⟩ : := DEF ⟨entity-type⟩ ⟨(entity-name)⟩ & ⟨statements⟩ ENDEF
4	⟨autonomous-def-block⟩ : := DEF ⟨(name)⟩ ⟨statements [stats⟩ ENDEF
5	⟨statements⟩ : ⟨attributes⟩ /& ⟨structural-description⟩ NOTE = ⟨attributes⟩ depend on ⟨entity-type⟩.
6	⟨structural-description [struct-descr⟩ : := ⟨PART-clause⟩ /& ⟨nested-def-block⟩ /& ⟨RELATED-clause⟩ /& ⟨INVOLVED-clause⟩ /& ⟨INVOLVED-IN-clause⟩ /& (⟨CONTAINED-IN-clause⟩ / ⟨DERIVED-FROM-clause⟩)/& ⟨LAYOUT-clause⟩/& ⟨IPO-clause⟩ NOTE = ⟨struct-descr⟩ includes also local ⟨structural-list [str-list⟩ and ⟨unstructural-list [list⟩ declared by clauses: ⟨str-list+⟩ (+name⟩ : : ⟨(list-of-components)⟩, ⟨list-name⟩ : ⟨(list)⟩.
7	⟨PART-clause⟩ : := PART: (part-name)/PARTS: (list-of-names)/ PART OF ⟨entity-type⟩ ⟨(entity-name)⟩ : (part-name)/ PARTS OF ⟨entity-type⟩ ⟨entity-name)⟩ : (list) NOTE = ⟨PART/PARTS-clauses⟩ are used in ⟨primary-entity-def-blocks⟩ ⟨PART-OF/PARTS-OF-clauses⟩ are external clauses used in ⟨autonomous-blocks⟩, for each ⟨(part-name)⟩ or component of list should be written separate ⟨primary-entity-def-block⟩.
8	⟨nested-def-block⟩ : := DEF1 ⟨entity-type⟩ ⟨(entity-name)⟩ ⟨statements⟩ DEF2 ⟨(entity-name)⟩ ⟨statements⟩ DEF3 ⟨(entity-name)⟩ ⟨statements⟩ ENDEF3 ... ENDEF2 DEF2 ... ... ENDEF2 ENDEF1 NOTE = ⟨nested-def-block⟩ expresses relation PART/PART OF and precedence, i.e. it is equivalent to ⟨structural-list⟩ but includes directly ⟨def-blocks⟩; processor of SL creates separate entity for each component identified by qualified name.
9	⟨RELATED-clause⟩ : := ⟨entity-identifiers⟩ RELATED TO ⟨entity-identifiers⟩ WHEN ⟨conditional-statements⟩ BY ⟨linking-technique⟩ [TEMPORARILY FOR ⟨time-interval⟩/ TEMPORARILY UNTIL ⟨date⟩] NOTE = ⟨linking-technique⟩ means one of the following techniques: POINTER/POINTER-ARRAY/ VECTOR/BIT-TABLE/CROSS-FILE.
10	⟨INVOLVED-clause⟩ : := ⟨entity-type⟩ INVOLVED: ⟨(entity-names-list) NOTE: e.g. USERS INVOLVED: (list of users) written in ⟨primary-defblock⟩ of PROBLEM.
11	⟨INVOLVED-IN-clause⟩ : := [(entity-name)] INVOLVED IN ⟨entity-type⟩ : ⟨(list-of-entity-names)⟩ NOTE = (user-name) INVOLVED IN PROBLEMS: (list of problems).
12	⟨CONTAINED-IN-clause⟩ : := [⟨entity-identifier 1⟩] CONTAINED IN ⟨entity-identifier 2⟩ NOTE = ⟨entity-identifier1⟩ is optional in ⟨primary-entity-defblock⟩, ⟨entity-identifier2⟩ means "storage center name".
13	⟨DERIVED-FROM-clause⟩ : := [⟨entity-identifier1⟩] DERIVED FROM ⟨entity-identifier2⟩ NOTE = ⟨entity-identifier1⟩ is optional in ⟨primary-entity-def-block⟩, ⟨entity-identifier2⟩ means "primary storage center"; both entities have been stored independently.
14	⟨LAYOUT-clause⟩ : := ⟨physical-structure-of-data⟩ NOTE = ⟨physical-structure-of-data⟩ may be written in COBOL DATA Division convention.
15	⟨IPO-clause⟩ : := INPUT ⟨(entity-name)⟩ [⟨CONTAINED-IN-clause⟩ / ⟨DERIVED-FROM-clause⟩] & OPERATION ⟨(name)⟩ USING ⟨(tool-name)⟩ OUTPUT ⟨(entity-name)⟩ DIRECTED TO ⟨("receiver-name")⟩ NOTE = ⟨IPO-clause⟩ is used in PROCESS and PROGRAM description, INPUT may be a record name or event name, OUTPUT is a report-name as a rule.

Entity types in SL language

The list of entity types proposed in the SL specification covers a variety of entities involved in information system design. These entities may be grouped into the following classes:

1. problem class (ACTION, ALGORITHM, ACTIVITY, DECISION, DEPARTMENT, DOCUMENT, PROCESS, EVENT, FLOW, INTERACTION, LIFE-HISTORY, OBJECTIVE, PROBLEM, SITUATION, STATE, STREAM, USER);
2. structuring class (BINDER, CLUSTER, INTERFACE, INTERSECTION, PART, STRUCTURE, TRACE),
3. program resources (MODULE, PACKAGE, PROCEDURE, PROGRAM, TASK);
4. data class (DBASE, FILE, GRANULE, MESSAGE, RECORD, REPORT, RESPONSE, SET, TABLE, TEXT, VECTOR);
5. technology class (BEGIN, ECODE, END, FORM, FOLDER, MODIFIER, PCODE, QUERY, PARAMETER, POINTER, RUN, SCHEMA, SUBSCHEMA, SESSION);
6. documentation class (DOCUMENTATION, PROJECT, STANDARDS);
7. staff class (ANALYST, PROGRAMMER, DESIGNER);
8. tools class (COMPUTER, DBMS, LANGUAGE, OPERSYSTEM, DDICT, TOOL);
9. undefined objects class (ENTITY).

and for each version of entity, called INCARNATION, which became a self-contained entity (independent from the parent entity). Another advantage of the SL language is that most of the statements are entity independent, i.e. they can be used for all types of entity.

There are two basic identifiers of entity: entity type and entity name. If entity name means group name (e.g. record name) then to get a record occurrence the key (IDKEY) should be supplied. The notation of SL includes the following conventional signs:

::=	determiner of metalanguage statement,
::	determiner of structural list,
:	determiner of nonstructural list,
=	value determiner,
⟨ ⟩	delimiters of nonterminal text in language definition,
(...)	delimiting parentheses of list,
/	alternative usage (or),
/&	cumulative usage ("and" option),
[&]	optional usage, obligatory usage ("and"); default sign,
(name)	entity name (if appearing after entity type),
(xxx)	simple name,
(a, b, c, d)	qualified name (first name indicates the highest hierarchical element, the last one means the lowest element) necessary to identify components of an entity declared by PART clause,
(x, y, z)	list of elements,
(x[y)	renaming by synonym (both names are valid),
(x; y)	renaming by replacement (only the second name remains valid)
x	sign of commentary within a line,
–	notational connecting sign (used in non-terminal text),
+)	continuation mark at the end of line or at the end of interrupted text,
(+	continuation mark at the beginning of next line or at the beginning of second part of text.

The notational presentation of the SL syntax is shown in table 1.

Most of the types are not obligatory and can be introduced by the definer in definition blocks. Obligatory entities are those which are referenced in predefinition of the SL version (see example N° 0). Some entities should not be predefined. This concerns BEGIN and END entities which are to be defined in definition blocks only when they contain additional user information (standard BEGIN and END labels need not be described). Predefinition cannot be used for the undefined object class dedicated to additional entities named after the ENTITY word in the DEF statement.

Another entities, e.g. E CODE and P CODE (event code and process code), have been introduced to meet the requirements of the structured design oriented to event-driven processing (data-driven processing). In this mode of processing, transactions enter in random sequence and at random time as records of the "common" input stream, containing information events from various business activities. In order to be correctly recognized and distributed into data bases they should be equipped with the "guide-code" named ECODE. An examplary structure of this code is shown in fig. 1. It consists of the activity code, resource code, operation code, and event category code. A detailed description of one method of coding the business events is presented in [3]. This method is closely related to the cross table mentioned (FACT). Each information event has to be recognized by the DISTRIBUTOR task module which establishes the "territory" of an event within a data base and invokes procedures necessary to process data located in input record.

Structural description of the structured entities

The first and most significant feature of the SL language is a capability to define a variety of structures not restricted to conventional top-down decomposition. SL provides expressions necessary to formulate and store structured entities that can be specified in bottom-up manner as well as top-down.

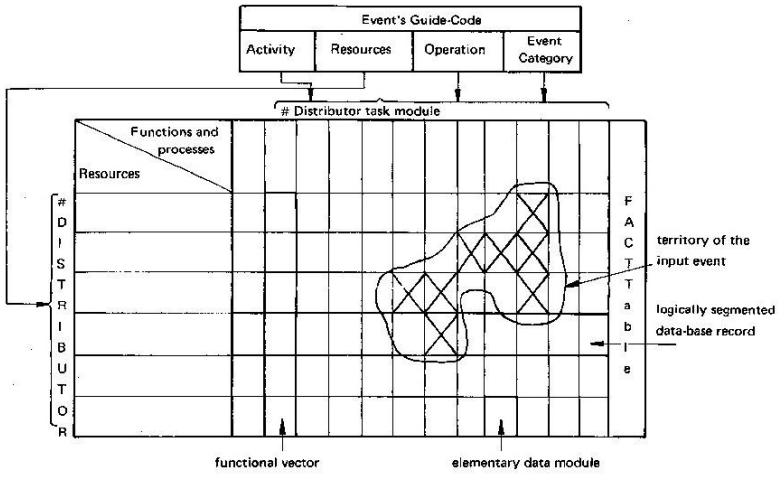


Fig. 1
Relationships between the FACT method and data-driven processing

As it was mentioned, SL statements dealing with structures may be located in blocks of the three following types:

a) predefinition block
used for definition of structural dependencies between entity types (e.g. STRUCTURE, BINDER);

b) primary-entity definition block
dedicated to describe attributes and structures known at the moment when the entity is set up;

c) autonomous block
used for complementary definition concerning any entity in any moment.

There are many structuring facilities provided by SL language:

1. key-structures (determined by key-words: STRUCTURE, BINDER, CLUSTER);
2. internal structural list
This list is named by the definer directly in the form (structural-list-name) :: (list of components) or can be associated with a secondary entity, e.g. DECISIONS :: (list of decisions).
An internal structural list does not imply the creation of a separate entity for each list, i.e. it has documentary meaning only. A list of components may be hierarchically decomposed using brackets, e.g. (uuu) :: (y(z, t), a(b, c)) which means: entity "uuu"' consists of parts "y" and "a", "y" consists of "z" and "t", parts of "a" are "b" and "c". A structural list determines completeness and precedence of components.
3. internal nonstructural list (lateral one level decomposition)
The naming of this list is made upon the same rules as for an internal structural list and the ":" determiner is used. No entity is created. No hierarchy and no precedence can be expressed within a list.
4. nested definition block (top down decomposition)
For each definition a block separate entity is created. Names of subordinated blocks have been qualified by the SL processor. Completeness and precedence should be provided in the specification. An optional entity (block) has to be equipped with an OPTIONAL attribute. A parallel one should be mentioned with the PARALLEL TO ⟨entity-identifier⟩ clause.
5. piecemeal decomposition
This structuring is used on any level of decomposition (or for any component) to make additional fragmentation. For example, it is applied when a predefined structure is not sufficient for a given entity, and using PART/PART OF clauses can provide required depth of decomposition.
6. environmental list (bottom-up)
This list specifies elements not being physical components of the described entity, but only appearing in its environment. To such elements there belong secondary entities specified in the INVOLVED/INVOLVED IN clause.
7. structuring in a space
By means of the LAYOUT statement a sequence of elements may be specified in terms of space (e.g. data items in storage).
8. operational structure
It is recognized by the IPO statement and expresses operational dependency between input and output.
9. logical structure
By use of the RELATED statement there can be created network data sets (in terms of DBMS) based on logical relations between records.

Fig 1a An Example of Cross-Tables for a Manufacturing Company

FUNCTIONS and PROCESSES / RESOURCES		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
				Short-term planning				Realization															
		Provision of codes and descriptions	Forecasting and long-term planning	Purchases	Sales	Manufacturing	Inventory /stocks/	Purchases	Sales	Manufacturing	Inventory control	Costing	Financial accounting	Repairing of equipment	Research and development	Administrative activity	Marketing						
1	Financial assets																						
	1. Share capital																						
	2. Operating profit																						
	.....																						
	8. Cash																						
	9. Loans																						
2	Raw materials																						
3	Components and finished goods																						
.	.....																						
5	Work in progress																						
6	Power and water																						
7	Equipment																						
	1. Manufacturing equipment																						
	2. Transportation equipment																						
	3.																						
8	Personnel and labour																						
9	Intangible assets																						
	1. Patents																						
	2. Copyrights																						

Table 2. List of entity types

ACTION logically closed group of events (e.g. received order acceptance)

ALGORITHM rules of action

ACTIVITY logically closed group of actions (e.g. scale)

ANALYST person professionally involved in problem analysis; analyst's name may appear in definition block of problems as a name of DEFINER

BEGIN header label of specification, equipped with passwords and user information, i.e. it contains control information which should be compared against the header label of specification batch having the same name

BINDER collection of entities belonging to various entity types

CLUSTER collection of entities belonging to the same type of entity

COMPUTER description of computers used in problem packages

DBASE data base (collection of data sets and entity descriptions)

DBMS data base management system

DDICT data dictionary

DECISION complex decision which needs to be described as an entity

DEPARTMENT organisational unit (department) involved in problems

DESIGNER person professionally involved in system design

DOCUMENT carrier of information event (source document)

DOCUMENTATION system documentation

ECODE event code (necessary to recognize an information event)

END trailer of specification (pattern and control information)

ENTITY user-defined entity

EVENT information event (reflection of business event)

FILE collection of data considered by the operating system as a whole

FLOW sequence of events at a given interval (period)

FORM table of parameters necessary to produce reports (headings, line-size, totals, page-size, etc.)

FOLDER arrangement of reports grouped together by identifying or other criteria (e.g. pertaining to one employee)

GRANULE portion of data and control information (in distributed processing)

INCARNATION version of entity

INTERFACE external connector

INTERSECTION list of entities involved in overlapped structures

INTERACTION interaction between processes

LANGUAGE programming language used to create a program resources applied in problem packages

LIFE-HISTORY diary of entity (sequence of states or events related to a given entity)

MESSAGE short output information

MODIFIER element of the body which should be modified in the preprocessing stage by the tuner

MODULE logically closed component of the program which can be invoked by a CALL statement

OBJECTIVE complex objective (optional entity used in problem description)

OPER-SYSTEM computer operating system under which a problem package may be put into operation

QUERY user's query to be stored

PACKAGE problem oriented package

PARAMETER control data items entered in execution time

PART component of the entity (equipped with qualified name)

POINTER form of an address pointer used in RELATED statement

PROCESS logically closed sequence of actions and events

PCODE process code (code necessary to recognize the process)

PROBLEM problem being the subject of description (the highest unit of description that represents usually a collection of logically related activities and processes; the relation may be based on the same objectives)

PROCEDURE portion of source code dedicated to some function or operation, and containing modifiers as a rule

PROGRAM executable program unit

PROGRAMMER person being the author of program resources; programmer's description includes knowledge and experience, the vital part of which is a list of program resources developed by himself

PROJECT collection of analysis and design documentations

RECORD component of the file, data portion considered as a whole by READ or WRITE instructions

REPORT batch output

RESPONSE interactive output

RUN execution of program

SCHEMA data base schema

SESSION batch or logically closed portion of D & L statements

SET logical collection of related records

SITUATION collection of states of logically related records at a given moment

STANDARDS standards for systems analysis and design

STATE state of a given entity (when it has to be stored as a separate entity)

STREAM input events stream (in event-driven processing)

STRUCTURE key-structure predefined at SL version level

SUBSCHEMA data base subschema (description of modifications and restrictions in comparison with SCHEMA, rules of invoking)

TABLE portion of data presented as a table

TASK piece of utility software

TEXT any text to be treated as an unformatted entity (black box)

TOOL computer aided design software and other tools

TRACE list of traces of a given entity occurrence (mapping of all relationships)

USER person having rights to use the package

VECTOR data vector or link vector (in a specific DBMS like ROBOT)

10. precedence relation
    This relation is used to set up a control structure that can be different from the physical decomposition. The PRECEDENCE OF statement may act also as a complementary clause to a nonstructural list.
11. origin relation
    It indicates the primary source of element by use of the DERIVED FROM statement or the actual source (CONTAINED IN statements).
12. trace relation (cross-reference)
    A tracing information is kept in the TRACE entity. It includes a list of all entities related to a given entity (named after TRACE word) or contains a collection of structural lists, sets, etc. which pass through the entity. Cross-references are to be maintained automatically for entities having the TRACE definition block and printed upon EXIBIT TRACE statement in & DL language.
13. overlapped structures ("common territory")
    In the definition block of an INTERSECTION entity there are listed structures which are under control.
14. structural inversion
    Conversions top-down into bottom-up (and vice versa) are made by using information stored in TRACE and INTERSECTION entities. It facilitates navigation through complex structures.
15. time relation
    A structuring in time is provided by STATE, SITUATION, HISTORY-LIFE entities (related to PROBLEM, ACTIVITY, PROCESS, ACTION, FILE, RECORD) and STREAM, FLOW entities (related to EVENT). Temporary links may be declared in RELATED statements.

The ways of structuring discussed above are focused on reflection of a natural flow of events happening in the real world, and an expression of changeable relationships between entities.

Examples of entity types description

Due to limited volume of this paper, only descriptions of some entity types have been presented.

Example N° 0. Predefinition of the language

```
BEGIN (name)
  DEFSL (version-name)
    DEF PROBLEM
      STRUCTURE :: (ACTIVITY (PROCESS (ACTION
      (EVENT))))
      SECONDARY ENTITIES INVOLVED: DEPART-
        MENT, ANALYST, USER, ALGORITHM, DECI-
        SION, OBJECTIVE, TOOL, DOCUMENTATION)
      ATTRIBUTES: (REQUIREMENTS, CLASS, DE-
        FINER)
    ENDEF
    DEF PROCESS
      PROCESS/ACTION :: (TRIGGER, ..., TERMINAL)
      SECONDARY ENTITIES INVOLVED: (PCODE,
        EVENT, ECODE)
      OBLIGATORY RELATIONS: (PRECEDENCE, IPO)
    ENDEF
    DEF PROGRAM
      STRUCTURE :: (MODULE (PROCEDURE))
      OBLIGATORY ATTRIBUTES: (LANGUAGE, COM+)
        (+PILER, OVER-SYSTEM, COMPUTER, PRO+)
          (+GRAMMER)
      OBLIGATORY RELATIONS: (PRECEDENCE-OF-
        PARAMETERS, PRECEDENCE-OF-MODULES)
    ENDEF
    DEF SET
      OBLIGATORY ATTRIBUTES: (TYPE, DBASE,
        DBMS)
      OBLIGATORY RELATIONS: RELATED
    ENDEF
    DEF ECODE
      STRUCTURE :: (ACTIVITY-CODE, RESOURCES-
        CODE, OPERATION-CODE, EVENT-CA+)
          (+ TEGORY-CODE)
      NOTE = components of the STRUCTURE OF ECODE
        should be specified as names after ENTITY word
          in definition blocks
      OBLIGATORY RELATIONS: LAYOUT location in
        EVENT
    ENDEF
    ...
  ENDEFSEL
END
```

Example N° 1. Problem definition

```
DEF1 PROBLEM (name)
[FNAME = full name of the problem]
DEFINER = analyst's name
CLASS = classification code of the problem
PRIORITY = priority code within the class
REQUIREMENTS = specific requirements of the problem
  have to be considered in information system design)
DEPARTMENTS INVOLVED: list of departments involved
  in the problem
ANALYSTS INVOLVED: list of problem analysts
USERS INVOLVED: list of users
ALGORITHMS INVOLVED: list of algorithms
ATTRIBUTES: list of attributes defined by the user or de-
  finer
TOOLS INVOLVED: list of tools used in problem analysis
DECISIONS: (list of decisions)
OBJECTIVES: (list of objectives)
DOCUMENTATION = name of documentation containing
  results of analysis
ACTIVITIES :: (aaal, aaa2, aaa3, ..., aaan)
DEF 2 ACTIVITY (aaal)
... attributes of activity
PARTS :: (bbb1, bbb2, bbb3, ..., bbbn)
  DEF 3 (bbb1)
  ...
  ENDEF 3
  DEF 3 (bbb2)
  ...
  PROCESSES OF (bbb2) :: (cc21, cc22, ..., cc2n)
    NOTE = relation ⟨entity-type⟩ OF ⟨(entity-name)⟩
      should be interpreted according to the statement
        STRUCTURE in DEF PROBLEM – see the
        example N° 0
  ACTIONS OF (cc21) :: (d211, d212, ..., d21n)
  EVENTS OF (d211) :: (eee1, eee2, ..., eeen)
  ENDEF 3
ENDEF 2
DEF 2 ACTIVITY (aaa2)
...
ENFEF 2
ENDEF 1
```

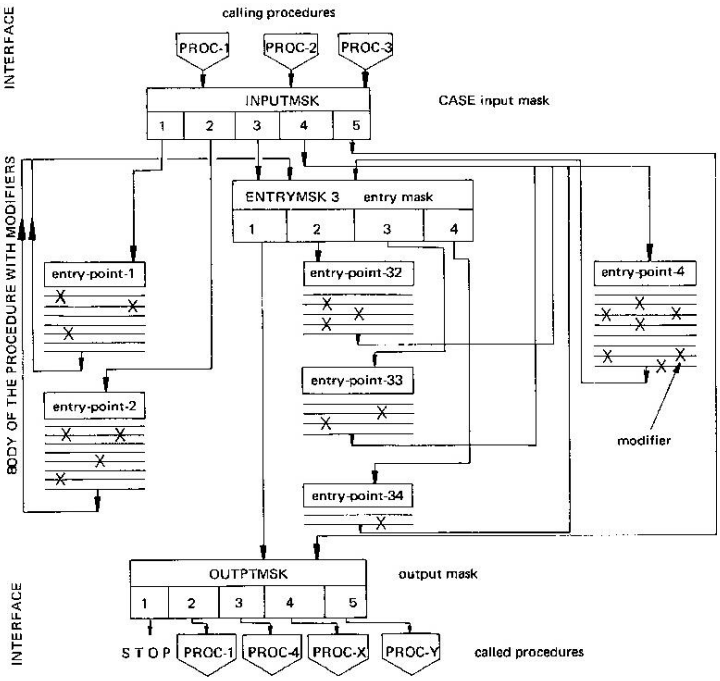


Fig 2 Complex Procedure Scheme

Example N° 2. Program definition

```
DEF1 PROGRAM (name)
   LANGUAGE = name
   COMPILER = name
   PROGRAMMER :: (chief-programmer-name (programmer-
      name))
   COMPUTER = name
   OVER-SYSTEM = (operating-system-name1/operating-
      system-name2)
   INVOLVED IN PACKAGE: (list of packages where this
      program is used)
   PRECEDENCE OF PARAMETERS: (list of parameters)
   PRECEDENCE OF MODULES: (list of modules)
DEF 2 MODULE (name)
   ...
   DEF3 PROCEDURE (name)
      MODIFIERS: (list of modifiers)
   ENDEF 3
   INPUT (name) DERIVED FROM/CONTAINED IN
      (name)
   OPERATION (name) USING (tool-name)
   OUTPUT (name) DIRECTED TO (name)
ENDEF 2
ENDEF 1
```

Concluding remarks

The specification language discussed here is to be considered as part of the specific structured design methodology dealing not only with "well" (hierarchically) structured problems and systems built under a long-term schedule of integration. A real cause for computerisation should be the actual business situation (there must be a man who wants information for decisions). The basis for such a structured design is some initial state of system, called pre-system, containing fundamental technological elements built according to the rules that enable the use of them in many problem packages. In this context special attention should be paid to modifiers, because they are the bottle-neck in making adaptive programs. There is a need to develop a specific technique to help programmers in writing so-called skeleton programs that have standardized control flow and are equipped with many modifiers (see fig. 2), which have to be processed by the tuner. The tuner may perform the following operations: insertion of data names and data values, renaming and redefining data, setting up the case (the input mask), renaming entry points, generation of CALL statements, generation of empty modules (driver or stub type), setting up interfaces between languages, insertions of expressions, invoking schemas or subschemas, etc. the final conclusion is that the proposed S & DL language seems to be a good initiative to improve the structured design centered on events, processes, and problems.

References

[1] *Stevens, W. P., Meyers, G. J., Constantine, L. L.*: Structured design. IBM Syst. Journal 2/1974

[2] *Yourdon, E., Constantine, L. L.*: Structured design. Prentice Hall, 1979

[3] *Ryznar, Z.*: A conceptual model of an interfunctional data base system. Information and Management 2/78

[4] *Couger, J. D., Knapp, R. W.* (edit.): System Analysis Techniques. J. Wiley & Sons, 1974

[5] *Wedekind, H.*: On the parametric specification of data base oriented information system. Management Datamatics 5/76

[6] A progress report on the activities of the CODASYL end user facility task group. Management Datamatics 5/76

Zweiteingang am 27.4.1981