Computer and Automation Institute,
Hungarian Academy of Sciences

COMPUTATIONAL LINGUISTICS
AND
COMPUTER LANGUAGES

XI.

Budapest, 1976.

Distributor for: Albania, Bulgaria, China, Cuba, Czechoslovakia, German Democratic Republic, Korean People's Republic, Mongolia, Poland, Roumania, U.S.S.R., People's Republic of Viet-Nam, Yugoslavia

K U L T U R A

Hungarian Trading Co. for Books and Newspapers
1389 Budapest,
P.O.B. 149, Hungary

For all other countries:

JOHN BENJAMINS B.V.
Periodical Trade
Amsteldijk 44
Amsterdam, Holland

Responsible Publisher:

Prof.Dr.T.VÁMOS
Director of the Computer and Automation
Institute, Hungarian Academy of Sciences

ISBN 963 311 039 4

Országos Müszaki Könyvtár és Dokumentációs Központ
házi sokszorositója
F.v.:Janoch Gyula

Contents

ON A PROPERTY OF THE CATEGORY OF PARTIAL ALGEBRAS

I.Németi

Mathematical Institute of the
Hungarian Academy of Sciences
Budapest, Hungary

In the study of semantics of programming- and other languages universal algebra and model theory has been playing an increasingly important role.

This seems to be natural, since generalised model theory is nothing but the mathematical representation of the so-called "possible worlds"-semantics which is the thorough approach to the problem of taking into account the existence of an external world and an internal mind, the sentences formulated in which they do have a meaning in /or refer to/ the external world. However, in this study, only those chapters have reached real maturity, which deal with models or algebras having total funcitons.

In the same time it has turned out that in computer science we cannot live without partial funcitons. Also in the semantics of natural languages "logic of actions" c.f. Hayes [1] would provide a better understanding and more adequate model theory than classical first order logic. But then again partial functions emerge. By now, many results have been reached by using total-function-model-theory, and therefore time is ripe to refine our tools. Many researchers have tried to "totalise" their partial functions c.f. Hayes' version of logic of actions or that of the Prague school. But as we know it only too well from the theory of algorithms, actions cannot be defined everywhere /at least if they are not severely restricted/. In program-language-semantics Burstall tried to use

total algebra by introducing a new element called "undefined".

These attempts have been really fruitful but they do have their limitations because algebras are just basically different. The category of partial algebras has many characteristic features which simply do not exist in any quasi-variety of total algebras. E.g. the class of strong epimorphisms coincides with the class of onto homomorphisms in any quasi-variety of total algebras; or there are the closed morphisms. They, however, do sometimes have something in parallel.

This paper is about a specific kind of morphisms of partial algebras the so-called closed morphisms c.f. Höft [3], and investigates this purely category theoretical concept c.f. Pásztor [4] in the variety of distributive lattices.

From now on $\mathcal{C}$ is an arbitrary category.

DEFINITION

A morphism $f \in \mathrm{Mor}\,\mathcal{C}$ is called *closed* if for any $gh=f$, if g is a bimorphism, then it is an isomorphism.

THEOREM

A morphism of partial algebras is closed in the above category theoretical sense iff it is a closed homomorphism /in the sense of e.g.Höft/.
A proof of this can be found in Pásztor [4].

Now, we give a characterisation of closed morphisms in the category $\mathcal{D}$ of distributive lattices. We use the terminology of Grätzer [5].

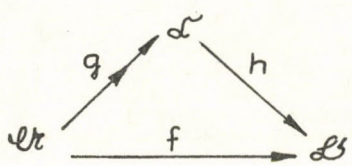
THEOREM

Let $\mathcal{A}$, $\mathcal{B}$ be distributive lattices and $\mathcal{A} \xrightarrow{f} \mathcal{B}$ be a homomorphism. Now, f is a closed morphism of $\mathfrak{D}$ iff the following are satisfied.

For any $a,x,b \in A$ such that $x \in [a,b]$ if $f(x)$ has a relative complement in the interval $[f(a),f(b)]$ in $\mathcal{B}$, then also x has a relative complement in the interval $[a,b]$ in $\mathcal{A}$.

PROOF:

1/ Suppose, the above condition is satisfied and



commutes while g is a bimorphism. Since $\mathfrak{D}$ is a variety, g is one-one. According to Grätzer's characterisation of epimorphisms /see Theorem 4, of chapter 13 in [5], pp.141/ the range of g /Rg g/ generates C by relative complementation, meet and join. Roman capitals stand for universes of the algebras denoted by the corresponding gothic capitals. Suppose $x \in [a,b]$ in $\mathcal{A}$. Then $g(x) \in [g(a),g(b)]$ in $\mathcal{C}$. Now, suppose g(x) has a relative complement in $[g(a),g(b)]$ in $\mathcal{C}$. Then $h(g(x))=f\,x$ again has a complement in $[f(a),f(b)]$ and therefore by hypothesis x has a complement in $[a,b]$ in $\mathcal{A}$. This proves that Rg g is closed w.r.t. relative complementation and therefore Rg g = C i.e. g is an isomorphism.

2/ Suppose that the "lattice-theoretical" conditions are not satisfied by f. We prove that f is not closed in the category theoretical sense.

Let $\mathcal{U} \xrightarrow{f} \mathcal{L}$, and while $x \in [a,b]$ does not have a complement in $[a,b]$ let $f(x)$ have a complement in $[f(a),f(b)]$. Let $y \in A$. Let H be the set of all mappings $AU\{y\} \xrightarrow{g} Ng$ such that $\mathcal{K}_g \in \mathcal{D}$, g is a homomorphism from $\mathcal{U}$ into $\mathcal{K}_g$ and $g(y)$ is the relative complement of $g(x)$ in $[g(a),g(b)]$ in $\mathcal{K}_g$. Form the product $\prod_{g\in H} \mathcal{K}_g$ /not minding set theory by co-well-poweredness/. Let $\mathcal{L}$ be the sublattice generated by the diagonal map $d = \langle H \cdot \{x\} \rangle_{x \in AU\{y\}}$ of $AU\{y\}$ into $\prod_{g\in H} \mathcal{K}_g$. Clearly $\mathcal{L}$ is a distributive lattice since $\mathcal{D}$ is closed w.r.t. products and subalgebras. Obviously, d is a homomorphism of $\mathcal{U}$ into $\mathcal{L}$.

We show that d is an epimorphism. Clearly $d(x) \cdot d(y) = d(a)$ and $d(x)+d(y)=d(b)$ because this holds in every projection of the direct product /if e_g is the g-th projection, then $d \circ e_g = g$/.

Now, by Grätzer's characterisation of epimorphisms, d is one.

Now, we show that d is a monomorphism, i.e. that d is one-one on A. Let $w,z \in A$ be arbitrary. It is well known that there is a homomorphism of $\mathcal{U}$ into the two-element lattice such that $h(w) \neq h(z)$. Extend h to $AU\{y\}$ by requiring that $h(y)$ be the relative complement of $h(x)$ in $[h(a),h(b)]$ in the two-element lattice. By this, d is one-one on A.

Thus d is a bimorphism of $\mathcal{K}$ into $\mathcal{L}$ but it is not an isomorphism, since x has no relative complement in $\mathcal{K}$ while d(x) has in $\mathcal{L}$.

Since $f \in H$, we have $f = d \circ e$ for some projection e of the direct product. Thus f is not closed.
Q.E.D.

THEOREM

Every morphism of the category $\mathcal{D}$ has a bimorphism-closed morphism factorisation.

PROOF:

The idea of the proof is to iterate ω times the above construction. In a single step we construct relative complements to every such $a \leq x \leq b$ for which f(x) has a relative complement in $[f(a), f(b)]$. The increased number of relative complements causes no trouble.
Q.E.D.

Keeping in mind the characterisation of closed and epimorphisms of partial algebras, it is interesting for comparison that:

PROPOSITION

Consider the $(H_s S_s P)$ variety of partial algebras /see[2]/ defined by the $(H_s S_s P)$ identities:

$$\exists\ c(y,x,z) \longrightarrow c(y,x,z) \cdot y = x$$
$$\exists\ c(y,x,z) \longrightarrow c(y,x,z) + y = z$$
$$c(y, x.y, x+y) = x$$

together with the identities defining the variety of lattices.

The class of distributive lattices coincides with the -reduct of this variety.

REFERENCES

[1] Hayes,P.: Logic of action, Machine Intelligence

[2] Andreka,H., Nemeti,I.: Generalisation of variety and quasivariety-concept to partial algebras through category theory, Preprint of the Math. Inst. Hung. A.S. 5/1976.

[3] Höft,H.: Operators on classes of partial algebras, Algebra Universalis 1972/2.

[4] Pasztor,A.: Closed morphisms in the category of partial algebras, Preprint 1976.

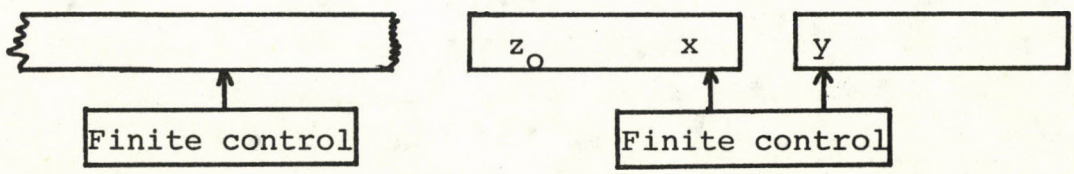
[5] Grätzer,G.: Lattice theory, Freemann and co. 1971.

A NOTE ON THE RELATION OF TURING MACHINES TO PHRASE STRUCTURE GRAMMARS

Gy. Rèvèsz
Computer and Automation Institute,
Hungarian Academy of Sciences
Budapest, Hungary

The aim of this note is to provide a possibly self-explanatory formal proof to the classical theorem of the equivalence of Turing acceptors and phrase structure grammars where the case of linear-bounded acceptors and context-sensitive grammars forms a trivial subcase.

For this purpose we will use a specific model of abstract automata, namely the two-pushdown machine /see Fig.1/. It is easy to show that this model is equivalent to the standard model of the single-tape Turing machine. Actually the two-way infinite tape of a single-tape Turing machine can be cut up at the read-write head and the nonblank portions of the two parts can be stored in two pushdown stores. A right or left scan of the original tape would correspond to the symbol by symbol copying of the contents of one pushdown store into the other. The expansion of the nonblank portion of the original tape corresponds to the insertion of a new symbol into an empty pushdown store.



TURING MACHINE TWO-PUSHDOWN MACHINE

Fig.1

Now we give the formal definition of the two-pushdown machine.

DEFINITION

A nondeterministic two-pushdown machine is a sixtuple

$$M = (Z, \Sigma, K, z_o, q_o, F)$$

where

Z is a finite set of *tape symbols*,
$\Sigma \subset Z$ is a finite set of *input symbols*,
K is a finite set of *internal states*,
$z_o \in Z-\Sigma$ is the *left endmarker*,
$q_o \in K$ is the *initial state*
F is a mapping called *transition function*

which can be given as a finite set of rewriting rules. Each of these rewriting rules may have one of the forms

1. $xqy \rightarrow xzp$
2. $xqy \rightarrow pzy$
3. $xqy \rightarrow xpz$
4. $xqy \rightarrow pz$
5. $xqy \rightarrow xpzy$

where $x, y, z \in Z$, $p, q \in K$ and $z \neq z_o$.

These rewriting rules define the moves which change the configuration of M.

DEFINITION

A configuration of M is a word of the form XqY where $X, Y \in Z^*$ and $q \in K$./Here X and Y represent the contents of the two pushdown stores, q is the actual internal state and the two read-write heads are scanning the last symbol of X and the first symbol of Y, respectively./ Similar models were already used in many papers, e.g., in [1], [2] and [3].

The relation $\underset{M}{\Longrightarrow}$ between two configurations is defined on the

basis of the rewriting rules in the usual way, i.e., $X_1\ q_1\ Y_1 \underset{M}{\Longrightarrow} X_2\ q_2\ Y_2$ if and only if there is a rewriting rule in F whose application yields $X_2\ q_2\ Y_2$ from $X_1\ q_1\ Y_1$ in one step. The transitive and reflexive closure of $\underset{M}{\Longrightarrow}$ will be denoted by $\overset{*}{\underset{M}{\Longrightarrow}}$. /A more detailed definition can be found for example on page 6 in [4]./

DEFINITION

The language accepted by M with empty store is

$$L(M)=\{P\in\Sigma^* \mid z_o q_o P \overset{*}{\underset{M}{\Longrightarrow}} pz, \text{ for some } p\in K,\ z\in Z\}$$

This means that the two-pushdown machine M accepts the word P if there is a finite sequence of moves that change the initial configuration $z_o\ q_o\ P$ into a configuration containing only two symbols. There may exist, however, several other sequences of possible moves starting with the same initial configuration, since we are dealing with nondeterministic models, i.e. more than one rewriting rules may occur in F with the same left-hand side.

THEOREM 1. To every two-pushdown machine M there is a phrase structure grammar G such that L(G)=L(M) and if M is linearly bounded then G is context-sensitive.

PROOF. For a given two-pushdown machine $M=(Z,\Sigma,K,z_o,q_o,F)$ we construct the phrase structure grammar $G=(V_N,V_T,S,R)$ as follows.

Let

$$V_M=\{S\}\cup(Z-\Sigma)\cup\{Z\times K\times Z\},\quad V_T=\Sigma$$

and the set of rules R correspond to the "inverse" of F, namely

1. $x[zpu]\rightarrow[xqy]u\in R$ for all $u\in Z$, iff $xqy\rightarrow xzp\in F$,
2. $[upz]y\rightarrow u[xqy]\in R$ for all $u\in Z$, iff $xqy\rightarrow pzy\in F$,
3. $[xpz]\rightarrow[xqy]\in R$, iff $xqy\rightarrow xpz\in F$,
4. $[upz]\rightarrow u[xqy]\in R$ for all $u\in Z$, iff $xqy\rightarrow pz\in F$,

5. $[xpz]y \to [xqy] \in R$, iff $xqy \to xpzy \in F$.

In addition to these we include

6. $[z_o q_o x] \to x \in R$ for all $x \in \Sigma$

7. $S \to [z_o\ q\ y]$, iff $z_o\ q\ y \to pz \in F$ for some $p \in K$ and $z \in Z$.

As can be seen from the construction $S \overset{*}{\underset{G}{\Longrightarrow}} P$, iff $z_o q_o P \overset{*}{\underset{M}{\Longrightarrow}} pz$, therefore $L(G)=L(M)$.
/A derivation in the grammar G corresponds to a reverse sequence of moves of M./

Moreover, it can be observed that if M has no type 5 rules, i.e. M is linearly bounded, then G is context-sensitive, Q.e.d.

In order to establish the converse theorem we need a normal form theorem for phrase structure grammars. In the sequel A,B,C and D denote nonterminals, a denotes an arbitrary terminal symbol, while P and Q denote arbitrary words in $(V_N \cup V_T)^*$.

THEOREM 2. Every λ-free phrase structure language can be generated by some grammar whose rules are all of the form $A \to a$, $A \to B$, $A \to BC$, $AB \to AC$, $AB \to CB$ or $AB \to B$.

PROOF. Suppose we have a phrase structure grammar generating the given λ-free language. We can eliminate all λ-rules from this grammar by replacing each rule of the form $P \to \lambda$ by the rules $Px \to x$ and $xP \to x$ where x ranges over the set of all nonterminal and terminal symbols. Only the empty word λ would cause a trouble since a derivation of the form $S \overset{*}{\underset{G}{\Longrightarrow}} \lambda$ cannot be obtained without λ-rules. Thus, we can get a grammar where each rule is of the form $P \to Q$ with $Q \neq \lambda$.

Now, if a rule $P \to Q$ is length-increasing (more precisely nondecreasing, or in symbols $|P| \leq |Q|$) then, according to a normal form theorem due to Kuroda [5], it can be replaced by a set of rules of the form $A \to a$, $A \to B$, $A \to BC$, $AB \to AC$, and $AB \to CB$.

So we have to deal only with length-decreasing rules. It is clear that terminal symbols can be eliminated from these rules by introducing for each terminal symbol a a new nonterminal A_a and including the rule $A_a \to a$. On the other hand, a length-decreasing rule of the form

$$A_1 \ldots A_m \to B_1 \ldots B_n \quad (m>n>0)$$

can be replaced by the set of rules

$$\begin{array}{ll}
A_{m-1}A_m \to C_mD_m, & C_mD_m \to D_m, \\
A_{m-2}D_m \to C_{m-1}D_{m-1}, & C_{m-1}D_{m-1} \to D_{m-1} \\
\quad\vdots & \quad\vdots \\
A_nD_{n+2} \to C_{n+1} & C_{n+1}D_{n+1} \to D_nB_n \\
A_{n-1}D_n \to D_{n-1}B_{n-1} & \\
\quad\vdots & \\
A_1D_2 \to D_1B_1 & \\
D_1B_1 \to B_1 &
\end{array}$$

where $C_{n+1},\ldots,C_m$ and $D_1,\ldots,D_m$ are newly introduced nonterminal symbols. These rules are either of the form $AB \to B$ or length-nondecreasing which completes the proof.

THEOREM 3. To every λ-free phrase structure grammar G there is a two-pushdown machine M such that $L(M)=L(G)$ and if G is context-sensitive then M is linearly bounded.

PROOF. We may assume that the grammar $G=(V_N,V_T,S,R)$ generating the given language is in the normal form established in Theorem 2. The corresponding two-pushdown machine $M=(Z,\Sigma,K,z_o,q_o,F)$ will be defined such that $Z=\{z_o\}\cup V_N\cup V_T$ (with $z_o \notin V_N\cup V_T$), $\Sigma=V_T$, $K=\{q_o\}$ and F is as follows.

1. $xq_oy \to xq_oz \in F$ for all $x\in Z$, if $z \to y \in R$,

2. $xq_oy \rightarrow q_oz \in F$, iff $z \rightarrow xy \in R$,
3. $xq_oy \rightarrow xq_oz \in F$, if $xz \rightarrow xy \in R$,
4. $xq_oy \rightarrow q_ozy \in F$, if $zy \rightarrow xy \in R$,
5. $xq_oy \rightarrow xq_ozy \in F$ for all $x \in Z$, iff $zy \rightarrow y \in R$.

In addition to these we include

6. $xq_oy \rightarrow xyq_o \in F$ $\quad x,y \in Z$,
7. $xq_oy \rightarrow q_oxy \in F$ $\quad x \in Z-\{z_o\}$ and $y \in Z$,
8. $z_oq_oS \rightarrow q_oS \in F$.

Again it follows directly from the construction that L(M)=L(G). Further, if G is context-sensitive then M has no type 5 rules so it is linearly bounded. Q.e.d.

References

[1] Minsky, M.L.: Recursive unsolvability of Post's problem of 'Tag' and other topics in the theory of Turing machines, Annals of Math. 74 (1961), 437-455.

[2] Walters,D.A.:Deterministic context-sensitive languages, Part II.
Information and Control 17 (1970), 41-61.

[3] Loeckx, J.: The parsing of general phrase-structure grammars.
Information and Control 16 (1970), 443-464.

[4] Salomaa, A.: Formal Languages.
Academic Press, 1973.

[5] Kuroda, S.Y.: Classes of languages and linear-bounded automata.
Information and Control 7 (1964), 207-223.

A NEW PROGRAM OPTIMIZATION

P.B. Schneck

Institute for Space Studies
Goddard Space Flight Center, NASA
New York, USA

Abstract

The *forward dominator* relation is a mirror image /dual/ of the ubiquitous back dominator relation of compiler optimizations /Allen and Cocke, 1972; Lowry and Medlock, 1969; Schneck and Angel, 1973/. The forward dominator relation is used to identify a new set of common subexpressions, not found by traditional techniques.

Introduction

Classical techniques for the discovery of common subexpressions in program units have been confined to the case where the evaluation of a subexpression is checked against *previous* evaluations *which are currently available*. Two criteria are used to decide whether a value is currently available.

1/ Does a previously calculated expression represent the current expression?

2/ Is the expression on a path which is certain to occur prior to the duplicate appearance?

The first criterion assures that the expressions in question are common. The second criterion tests the "back dominator" relation to assure that the result of the first appearance of the common subexpression calculation is always

available when the later expression is reached.

The approach described in this paper handles common subexpressions occurring in circumstances which violate the second criterion. If a potential common subexpression *must be evaluated** it will be moved to a point which satisfies the second criterion and common subexpression analysis can then be performed by standard techniques.

COMMON SUBEXPRESSION ELIMINATION: THE CLASSICAL APPROACH

In this section of the paper, the path of evaluation of common subexpression elimination is examined. Early efforts were limited to a single statement, extended to a basic block, and further extended to entire programs. The back dominator relation is essential to the inter-statement optimizations. Limitations of this approach are shown.

Analysis Within a Statement

The simplest context for discovery and elimination of common subexpressions is a single statement. When formally identical subexpressions are found to exist, only the first evaluation needs to be performed. This approach was used in the Fortran I compiler /Sheridan, 1959/. An obvious limitation of the restriction to a single statement is the much smaller scope available for optimization. This scope can be expanded without disturbing the simplifying assumption of straight line program flow.

---

* If the expression is not evaluated along certain paths, then forcing its evaluation could lead to an error condition which would not otherwise occur. This is termed a violation of safety /Schaefer, 1973/.

Analysis in Regions with Straight Line Flow

A sequence of statements exhibiting straight line program flow with only one entry /at the beginning of the sequence/ and only one exit /at the end of the sequence/ is called a basic block. Recognition and elimination of common subexpressions within a basic block is similar to the processing within a single statement except that testing for formally identical common subexpressions is replaced by testing to determine that the same values participate within an expression. The "value numbering" scheme /Cocke and Schwartz, 1969/ is preferred because:

1/ Common subexpressions are not limited to formal identities

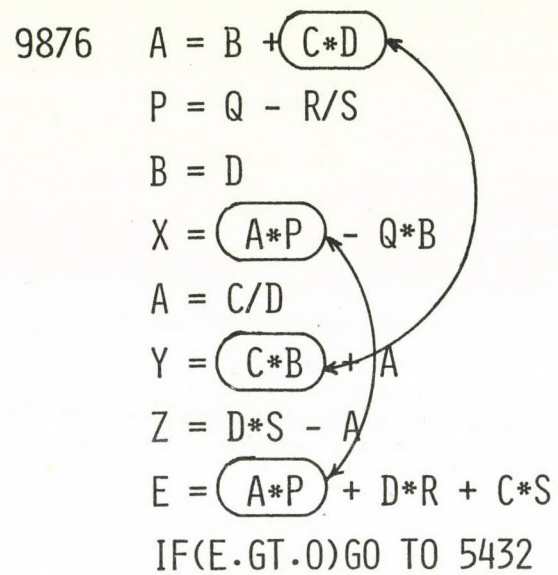
2/ Common subexpressions are known not to occur if a participating variable is modified between two formally identical subexpressions.

Figure 1 shows two common subexpressions occurring within a basic block. Definition of either "*A*" or "*P*" between the formally identical subexpressions ("*A***P*") would result in their treatment by the value number scheme as separate expressions. Additionally, the formally distinct subexpressions ("*C***D*", "*C***D*") are found to be common.

Because of the straight line flow within a basic block the first appearance of a common subexpression results in its availability at all subsequent statements.* When later appearances of a common subexpression are encountered the first evaluation must already have occurred. It is always possible to replace later evaluations of a common subexpression by the result of the first evaluation.

---

* Each statement is said to back dominate all successor statements in the basic block because it must be executed before they can be reached.



Each statement back dominates all of its successors within the basic block.

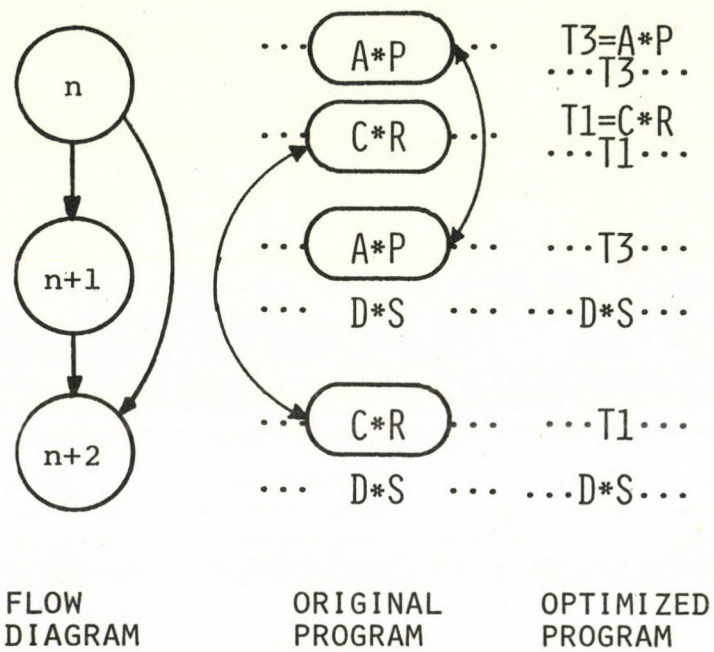
Fig.1 *Common subexpression recognition within a basic block*

Analysis in the Presence of Program Flow

The availability of a prior computation, which is taken for granted within a basic block, needs to be established when potentially common subexpressions do not occur in a single basic block. Figure 2 illustrates a simple example of "non-linear" program flow and its effect upon potential common subexpressions. Block n back dominates both block $n+1$ and block $n+2$ because it is necessary to traverse block n to reach either of those blocks. Block $n+1$ does not back dominate block $n+2$ because it is possible to reach block $n+2$ without traversing block $n+1$ /directly from block n/.

The back dominator relation assures that computations performed in block n are available for use in block $n+1$ and block $n+2$. Therefore the subexpressions $A*P$ appearing both block n and block $n+1$ are common. Similarly the subexpressions $C*R$ appearing in block n and block $n+2$ are also common. In each case the second appearance of the common subexpression may be replaced by the result of the calculation which appeared in the back dominator block.

Block $n+1$ does not back dominate block $n+2$, and so computations in block $n+1$ may not be available for use in block $n+2$. The path from block n to block $n+2$ makes it possible to skip block $n+1$ and any calculations it contains. The appearances of the subexpressions $D*S$ in block $n+1$ and block $n+2$ cannot guarantee a common subexpression /in this classical back dominator framework/ because the first appearance of $D*S$ may be skipped and therefore no result is available to replace the second appearance.



Block n back dominates block n+1 and block n+2. Common subexpressions are found when they occur both in one of those blocks (n+1, n+2) and its back dominator (n).

Fig.2 *Classical "back dominator" common subexpression recognition*

COMMON SUBEXPRESSION ELIMINATION: A NEW APPROACH

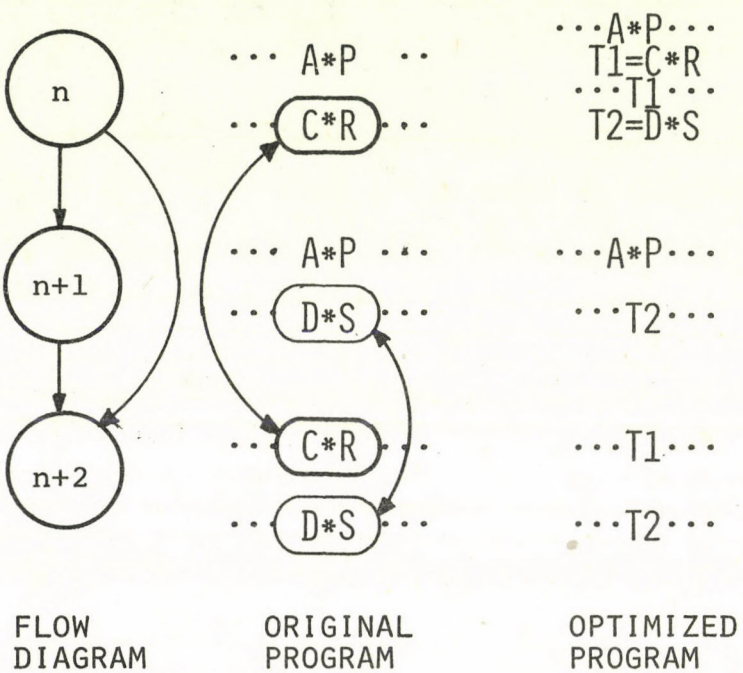
The back dominator relation is the foundation of the classical techniques for recognition of common subexpressions. In this section the dual relation, forward dominator, will be shown to be useful for the recognition of many additional common subexpressions.

The back dominator relation is true when a particular block, the back dominator, must be entered before a given block can be reached. The forward dominator relation is true when a particular block, the forward dominator, must be entered after a given block has been reached.

Overview

If a potential common subexpression occurs in some block and also in a forward dominator of the block then it may be possible to introduce a computation of the subexpression at a point which back dominates both the block and its forward dominator. /This point always exists because the entry block dominates all other blocks of a program./ If there are no definitions of the common subexpression's variables between its point of insertion and the first of the original subexpressions then the inserted subexpression calculates a value which can be used in lieu of the original common subexpressions.

Figure 3 illustrates the general approach as just discussed. Block $n+2$ forward dominates both block n and block $n+1$. The appearance of $C*R$ in block n and block $n+2$ means that $C*R$ can be treated as a common subexpression. /Because block n back dominates block $n+2$ this common subexpression is also found by the classical method./ Similarly the appearance of $D*S$ in block $n+1$ and block $n+2$ permits it to be treated as a common subexpression. The subexpression, $D*S$, is moved to block n, the common back dominator of block $n+1$ and block $n+2$. The two original appearances of $D*S$ are then replaced by the newly



Block n+2 forward dominates block n and block n+1. Common subexpressions are found when they occur both in one of those blocks (n, n+1) and its forward dominator (n+2). The evaluation of the common subexpression is moved to block n, the back dominator of block n+2 (and therefore the back dominator of block n+1).

Fig.3 *"Forward dominator" common subexpression recognition*

available value.

When forward and back dominator processing are performed consecutively all potential common subexpressions are discovered. Figure 4 illustrates the example program after it has been processed by both techniques.

The Algorithm

The blocks of a program are examined pairwise, until a pair is found where the second block forward dominates the first. If at the same time the first block back dominates the second, the pair is skipped because the classical /back dominator/ analysis will expose common subexpressions within the pair. Once a pair of blocks is selected a search is made to determine whether or not /based upon the hypothesis that the first block back dominates the second block/ there are any common subexpressions. When the following pair of conditions is met, any subexpressions which are found will be moved to a block which back dominates both blocks. First, no definition of the variables involved may occur between the desired point of insertion and the earlier* appearance of the common subexpression. This condition assures that the newly inserted definition is computed using the same values of the variables as in the original expressions. Figure 5 shows why the back dominator of the forward dominator must be used. Second, because of considerations of safety, the point chosen for insertion of the calculation of the common subexpression must be forward dominated by the forward dominator block of the pair. This second condition assures that the new evaluation will be performed only when the original common subexpression would have been evaluated. Figure 6 illustrates a violation of the requirement for safety and the consequent inability to perform the common subexpression eli-

---

* A topological order, such as is obtained by interval analysis, defines "earlier". This definition also avoids movement of expressions into inner loops, which would otherwise increase program time.

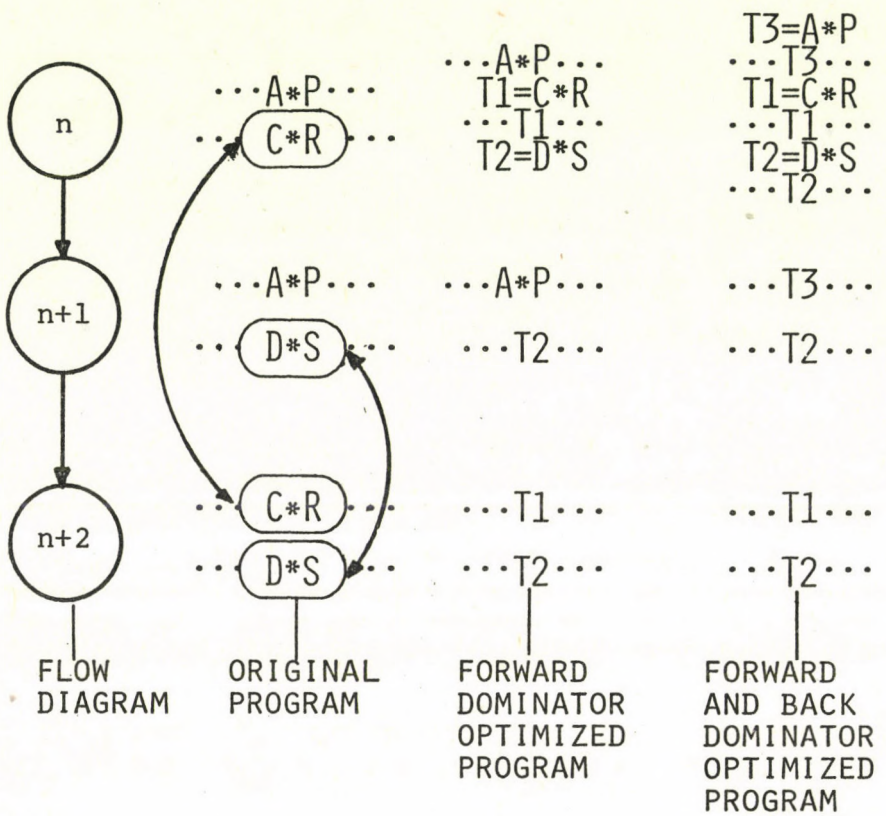
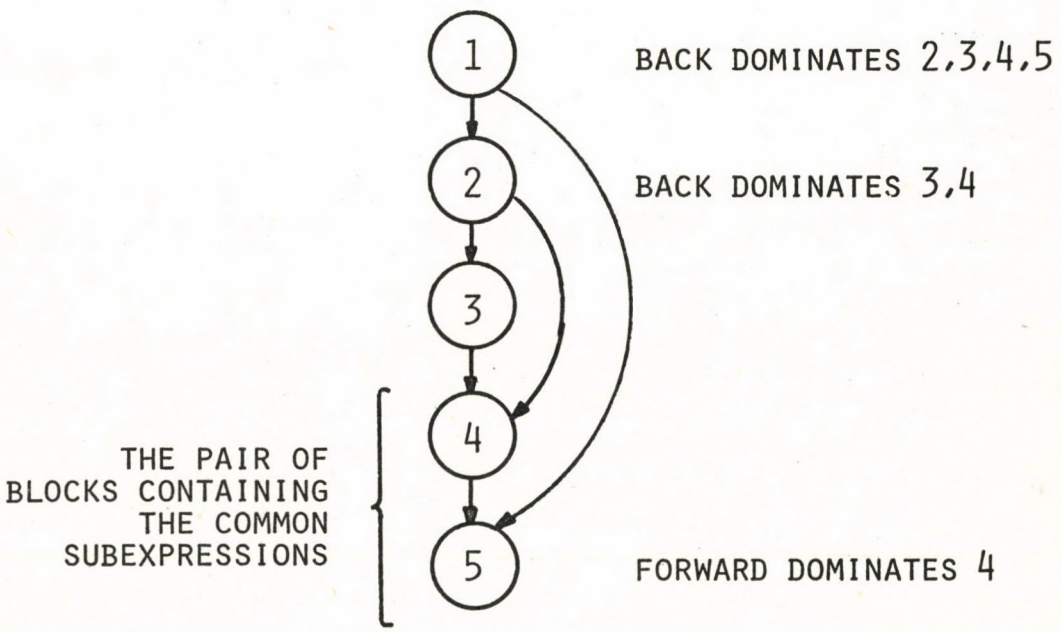
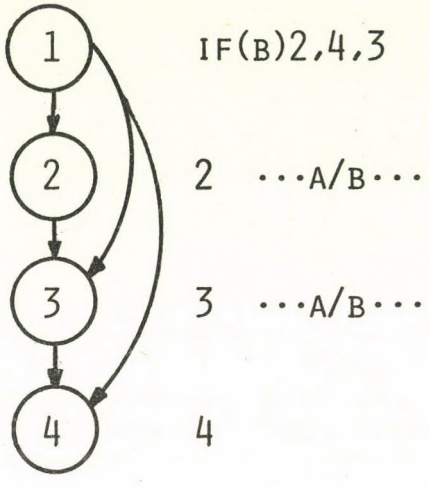


Fig.4 Forward dominator recognition, followed by back dominator recognition



The calculation point must back dominate both blocks of the pair containing the common subexpressions (4,5). The back dominator of block 4 is not a back dominator of the forward dominator of the pair. Block 1, back dominates both blocks of the pair.

Fig.5 *Selection of the point for calculation of the common subexpression value*



Blocks 2 and 3 comprise the pair containing the candidates for common subexpression elimination. The back dominator of block 3 is block 1. Movement to block 1 would result in the evaluation of A/B even when the branch to block 4 is taken, resulting in a division by zero that would not otherwise occur. Therefore common subexpression elimination is not performed.

Fig.6 *Common subexpression elimination inhibited by safety considerations*

mination.

After the calculation of the common subexpression is moved to the back dominator block the later appearances are replaced by the value obtained.

Effectiveness of the Forward Dominator Technique

As indicated earlier, the forward dominator and back dominator relations are dual. This is easily seen in the following definition:

Block i is said to back /forward/ dominate block j if all paths between the unique program entry /exit/ block and block j must contain block i.

Inverting the direction of flow reverses the roles of entry/exit and back/forward dominators. From a graph theoretic point of view the two methods are equally powerful. The forward dominator technique has an additional requirement - moving the calculation of the common subexpression to a back dominating block - which can inhibit its effectiveness. In order to move the common subexpression, the two requirements of 1/ no modifications to variables between the three blocks involved and 2/ safety, must be satisfied.

Common subexpressions may be categorized as belonging to one of three general classes:

1/ Satisfying both back and forward dominator relations.
2/ Satisfying a back dominator relation.
3/ Satisfying a forward dominator relation.

The first class is handled by either the back dominator or forward dominator technique. Preliminary investigation reveals that similar numbers of common subexpressions may be eliminated in the remaining two classes. Thus, introduction of this new class of common subexpression optimizations significantly increases the range of optimizations occurring.

Conclusion

A new technique is described for discovering and eliminating a new class of common subexpressions. It is shown to be as powerful as existing techniques for current classes of common subexpressions. An algorithm for the technique is discussed, and has been implemented within an operational compiler.

Acknowledgement

The author wishes to acknowledge the work of Nadim Habra, who implemented the forward dominator algorithm on the basis of a preliminary description.

References

[1] F.E.Allen and J.Cocke, Graph Theoretic Constructs for Program Control Flow Analysis, IBM Research Report.

[2] J.Cocke and J.T.Schwartz, Programming Languages and Their Compilers, New York University, 1969.

[3] E.S.Lowry and C.W.Medlock, Object Code Optimization, Communications of the ACM, Vol.12, Number 1, 1969.

[4] M.Schaefer, A Mathematical Theory of Global Program Optimization, Prentice-Hall, 1973.

[5] P.B.Schneck and E.Angel, A Fortran to Fortran Optimising Compiler, The Computer Journal, Vol.16, Number 4, 1973.

[6] P.B.Sheridan, The Arithmetic Translator Compiler of the IBM Fortran Automatic Coding System, Communications of the ACM, Vol.2, 1959.

FORMAL DESCRIPTION OF SOFTWARE COMPONENTS BY STRUCTURED ABSTRACT MODELS

B.Dömölki, E.Sánta-Tóth /Mrs/

SZÁMKI Research Institute for Applied Computer Sciences
/formerly INFELOR Systems Engineering Institute/
Budapest, Hungary

ABSTRACT

Structured Abstract Model /SAM/ is a description of some *object* in the form of a sequence of levels structured according to the hierarchy of design *decisions*. Description /or design/ of the object is given as an ordered set of "SAM-forms", each describing in a *well-defined structure* one - or several strongly connected - decisions, together with all their consequences. Decisions appear in the form of the definition of some of the *concepts* necessary to describe the object. This definition is given in terms of primitive concepts, not to be defined further on that level. Such a model can be *verified* by giving on each SAM-form our *assumptions* about the primitive concepts and proving the necessary properties of the concept/s/ to be defined on that level /provided that each assumption will be proved on the level, where the concept will be defined/.

Software components offer a class of objects very much suitable for such type of formal descriptions. In the paper the results of our three year research are reported, covering

- the investigation of *methodological* problems connected with SAM-like descriptions, including the application of these principles to develop a system to support program design and implementation;
- descriptions of abstract models for *real software*

objects /like assemblers, editors etc./, aimed as a first step towards creating a library of such models /"Software Encyclopedia"/.

INTRODUCTION

In recent years there has been a great increase in the number of application areas, methods and facilities in the computer field and at the same time in the number of non-professional programmers. This requires the development of a new /user-/ software environment in which communication with the computer is done not by programming in the traditional sense only, but partly or wholly by giving the specifications of the problem to be solved. The complexity of the specifications can be decreased by structuring them of our design decisions, allowing the stepwise refinement of concepts. A software system should be developed which allows its user to

- employ terms and *concepts* native to his own speciality,
- give *non-procedural problem definitions* by specifying relations among these terms,
- use *hierarchical problem specifications*,
- *verify* his decisions on all levels.

Theoretical computer science has produced several important results towards this goal in the fields of Mathematical Theory of Computation, Artificial Intelligence, Programming Methodology etc. On the other hand, modern *practical* methods of program design and implementation are beginning to be used successfully at some software development enterprises. The gap between theoretical research and practical results is a fact, widely recognized in the literature.

The research outlined in this paper is aimed to take an intermediate position between theory and practice, by studying /describing, verifying, classifying, etc./ concrete software objects with theoretically based abstract methods.

As a first step towards this goal we are interested in

finding methods for the formal description and verification of *abstract models of programs*. These methods can be used to develop a means of design and implementation which may help achieve a more exact and efficient form of traditional programming and which may also be a step in the transition toward the kind of new problem specification and programming mentioned above. With these methods it would be possible to *describe and discuss in a uniform manner* the software elements occuring in programming practice /assemblers, loaders, operating systems/. The lack of such descriptions has been realized by the designer and customer of the software product as well as by the educator of programmers.

The purpose of this paper is to give an overview of the research activity in this direction, initiated in our institute* in 1973 and materialized in the internal research reports and diploma thesises /written mostly in Hungarian/ listed in the Appendix.

In subsequent sections we shall give the definition of the subject /section 1/, examine the questions of *methodology* /section 2/ and give the results we have achieved so far in the description of software elements /section 3/. Then we shall summarize the application possibilities of the research of abstract models /section 4/.

References to published papers will be given by author and year of publication /e.g. [Dijkstra, 72]/, while internal papers listed in the Appendix will be referred by number /e.g. [3]/.

---

* Research Institute for Applied Computer Sciences /formerly INFELOR Systems Engineering Institute/, Budapest, Hungary

1. DEFINITION OF THE SUBJECT

The basic problem of the "Software Crisis" /[Boehm, 73]/ is the difference between the order of magnitude of the complexity level of the problems to be solved and that of objects directly comprehensible to machines used in their solution /see Fig.1a./. The *"complexity problems"* stemming from this difference can only be solved concurrently with the development of *programming methodology*. This gap can be bridged by introducing intermediate levels /see Fig.1b./. Here - using Dijkstra's analogy of a "necklace, strung from individual pearls" /[Dijkstra, 72]/ - each level /"pearl"/ in effect defines an *abstract machine* in virtue of the primitive concepts /i.e. operations and data structures/ used on that level. Concepts occuring as primitives on higher levels can be defined in terms of "programs" for this machine.* The "distance" between these levels - given that the definition of the levels is good - ought to be as small as to preclude the appearance of the complexity problems, Furthermore, exactly specifying the *primitives* at all levels the *correctness* of the link between the various levels can be ensured.

This actually means the following /see e.g. [Dijkstra, 72]/. If at any level you "cut" the necklace you can state: if there is an *abstract machine* whose machine objects are the primitive concepts unresolved above the cut, then the necklace portion above the cut can be viewed as a *program* for this machine /Fig.2, left side/.

Or looking at it in a little different, subsequently more useful manner: let P denote the set of those primitives that are referred to at levels above the cut, but are not defined there. We can then say that the portion above the cut consists of descriptions making use of the elements of P as primitive concepts, while the part below the cut contains the elements of P /see Fig.2 right side/.

---

* D.Varga has pointed out the similarity of these ideas to the natural language description methods proposed by P.Sgall.

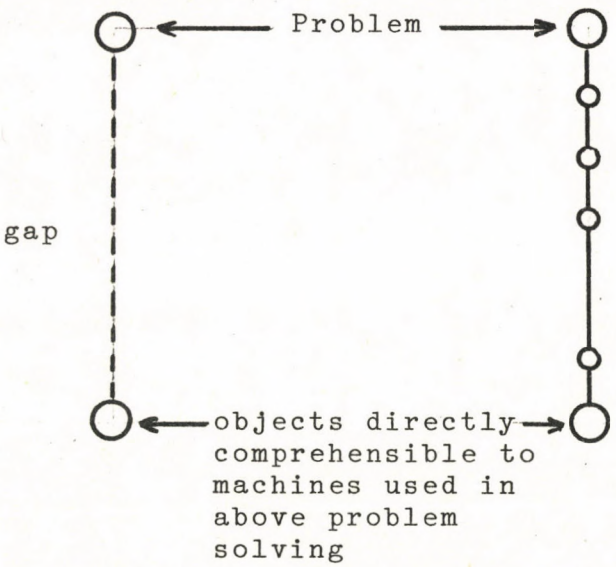


Fig.1a *The "complexity problem*

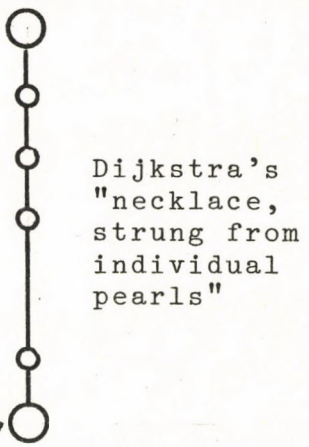


Fig.1b *Dijkstra's dissolution of complexity problem*

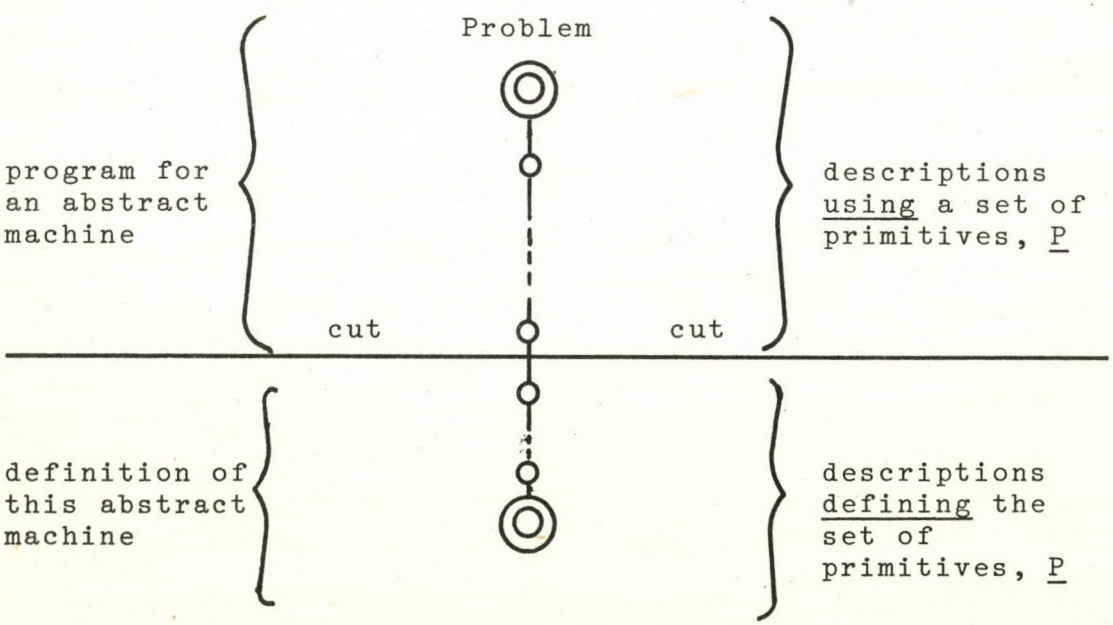


Fig.2 *Possible interpretations of Dijkstra's pearls*

In the top-down, structured view the problem solver starts from the definition of the problem and continues by *stepwise refinement* until the remaining primitive concepts are either

- known to some machine, in other words, these primitives are *implemented* on the given machine, or
- the machine can *synthesize* them from the specifications of the primitive concepts /i.e. from the requirements they are expected to fulfil/.

In traditional programming systems the "intelligence" of the machine materializes in the implementation of the concepts of some programming language; in future machines a formal description /which is "good" in the same /computer environments/ intelligence will manifest itself in the capability to synthetize concepts from their specifications /thus programming languages - as we understand them today - will become superfluous/.

Our research aims at the development of the formal methods of such a *top-down, structured, verifying program design* description. Our first application of this method [Dömölki,73] gives the description of an abstract assembler model. This paper saws a possible solution of the problem, in linking three known methods:

- starting from the descriptional method of *VDL* developed by the Vienna Laboratories of IBM /[Lucas, 68], [Neuhold, 71] [Lee, 72b], [Wegner, 72]/;
- following the principles of Dijkstra's *Structured Programming* /[Dijkstra, 70, 72], [Mills, 72]/;
- applying the axiomatic *program correctness verification method* proposed by Hoare /[Hoare, 69, 71a, 72a-b-c]/ to the programming language defined by the two previous points; we get a *descriptional method* which can be used to formally describe the *Structured Abstract Models /SAM-s/* of hierarchically ordered problem families in as much implementation and machine independent manner as possible.

2. METHODOLOGY

In the previous section the main ideas of SAM-research were summarized. This research enables the development of an - in some sense "good" - design and implementation method, and at the same time allows us to give a formal description /which is "good" in the same sense/ of software products. This section deals with the methods that can be used to make such descriptions.

The methodological goal of SAM-research is to establish a *means of problem elaboration that is top-down, abstract, directed by a well-structured hierarchical order of decisions and is verifiable.*

We started with VDL which turned out to be a good abstract description /design/ mechanism for compilers /see 3.3/. *Algorithm descriptions* given in VDL *were expanded with textual and verification parts.* The applicability of the method to the description of problem families was tested on practical problems /see [16]/.

We shall now examine the questions of methodology of description and verification of these models as well as the basic features of a Software Support System for SAM-like program design and some problems of the implementation of programs designed in VDL.

2.1 Description of models

The *design* and *implementation* of complex objects /e.g. computer programs/ is realized through a sequence of decisions. Determining the *correct order of* these decisions and describing their /immediate/ *effects independent of one another* can greatly enhance the efficiency and lucidity of the design.

By a *Structured Abstract Model /SAM/* we mean a description of some object. This description has distinct *levels* according to the decisions made during design; at each level one /exceptionally more than one/ decision and all its immediate

consequences are described. Each decision means the definition of a *concept* used in describing the object /e.g. in the case of programs a procedure- or data-structure/. The definition is made in terms of *primitive* concepts not defined further at the given level. Thus at every *SAM-level* we must give

- a *problem* definition, which defines the task of this level,
- a *decision*, which is usually the *definition/s/* of a concepts/s/ occuring as primitive at one of the previous levels,
- a *list of* new concepts used as *primitives* in this decision /definition/,
- the *specification* of the primitives /i.e. our hypotheses about them/, and

if we are also interested in verification,

- an *assertion* giving the properties of the concept defined on this level,
- some sort of proof /formal or not/ of the assertion as a *theorem*. This may require the statement of further hypotheses describing inter-statement relations: *lemmas*.

Thus the question, whether an object has a certain required property /in case of programs their *verification*/ can be reduced to

- the proving of the assertions about the properties of the concepts, to be performed independently for each level /using the hypotheses about the primitives of that level/, and
- the examination of whether the various levels have been properly combined that is to ensure that every specification and lemma is proved as a corresponding assertion at a subsequent level.

The concepts that are not defined at any level are called *primitives with respect to the whole model* and hypotheses applied to them are treated as *axioms*. /In case of programs these primitives, which form the bottom-level, can be the statements and standard procedures of the programming language used./ There is no limitation, however, to how deep we go in a

given model in refining the concepts we view as primitives. Thus models that are left "unfinished" at higher levels, in virtue of leaving open a number of design or implementation questions, determine a larger set, a family of *concrete objects*. In this way it is possible to describe sets of concrete objects ordered by design decisions.

While the above considerations define a rather strict *structure* of the description, we do not want to impose any limitation on the formalization of our language: any language can be used which allows unambiguous determination of the primitive concepts from the definitions.

Thus currently we give the description of our models in two parallel languages. On each level we give

1. a *textual*, natural language description, and
2. a *formal* description /at present in VDL*/,see Table 1.

Ad 1. The *textual description* discusses the question/s/ raised by the problem, using the textual definition of the problem as a basis. Each question is followed by a list of possible *solutions, alternatives*. This is followed by a *decision* which constitutes the factor determining the role of the SAM-level in the model. There may be *several decisions* applicable to a question; in this case models with different properties may be originated from the different decisions. /*Such model-families* can be represented by a *tree* - an example of this can be found in [16]. The *nodes* of this "tree-model" are the questions /or problems/, its *branches* the selected solutions based on the decisions. The latter generates the model corresponding to the subgraph defined by them./

A decision is followed by its *justification*, perhaps an *explanation*, and a list of *consequences*.

The correspondence between levels and decisions can be either.

---

* The possibility to use some other abstract program specification language instead of VDL is also considered, including the new Vienna technique for the description of semantics /see [Bekic, 74] and [18]/.

level .....

	textual /informal/ part	*formal part /VDL/*
description	- problem definition (which concept/s/ will be defined in this level) - possibilities or alternatives - decisions and consequences - list of new (primitive) concepts	- list of primitives (which will be defined in this level) - (family of definitions) - data- and procedure-definitions - list of new primitives
verification	- specification: hypotheses about the primitive concepts - assertions about concepts defined on this level	- specification: pre- and post-conditions for the primitive procedures - theorems: pre- and post-conditions for the procedures defined on this level
	- informal considerations about the validity of the assertions (as consequences of the hypotheses)	- formal proof of the theorems

Table 1. SAM-"form"

a/ such that each level contains one decision /or several decisions if they are strongly connected/, together with all their consequences; or

b/ such that each level contains the definements of all primitive concepts occuring on the immediately preceeding level /e.g. as in THE operating system, see [Dijkstra, 68]/.

There are no significant differences between these two approaches, since each b-type level can be substituted by a set of related a-type levels. For reasons of simplicity in the following we will use levels in the sense of a/.

There may be several questions raised on a given level and correspondingly several decisions, if these are connected in some way.

In the *specification section of the textual description* the *primitive* concepts occuring in the definitions /determined by the decisions/ must be *listed*, together with hypotheses about them /enumerating all the assumptions made about the concept in the definitions/, and if we are verifying we must prove the hypothesized properties of the concepts defined.

It is obvious that even if we examine only the above mentioned textual, informal sections of the SAM-forms we shall see a clear well-structured text; its reader can review the *steps of the problem solution* - essentially - without misunderstanding. That means, that if we organize the description according to the structure and principles described above, the "readability" and "structuredness" of our design can be improved even without introducing any formal language. The importance of this kind of description when several people are working on a *program design* is equally obvious.

Ad 2. On every level we also give a *formal* description in VDL /using the extensions proposed by [Lee, 72b]/. This formal description contains a VDL definition of the direct consequences of the decisions made in the textual part in the

form of definitions /or refinements/ of some of the data structures and procedures that occured as primitives at higher levels. The formal description consists of *data* and *procedure definitions* followed by a *list of primitives* used in these. An important factor is that if we refine a data-structure on this level then all of its accessing procedures should be refined accordingly on the same level.

An important requirement of the two /formal and textual/ language variants of the description - both covering all sections of the SAM-form - is that they should be related to each other in the following sense: there should be a one-to-one correspondance between the decisions of the textual section and the data and procedure definitions; the list of primitives should be comparable to the ones used in the textual description.

In the formal variant of the specification section we may list the hypotheses about the primitives, i.e. the requirements that the procedure primitives on this level are expected to fulfil */pre- and post-conditions/*. Again, there are no limitations on the language of these requirements except those already made for the text of this level i.e. the primitives used in them should be comparable /or identical/ with the list of primitives for this level.

2.2 Verification methods

If in the formal description of the SAM-form we gave the specifications, then in the *verification section* these are treated as hypotheses and the proofs of the assertions /theorems/ about the properties of the procedures defined on this level, are reduced to the proof of the hypotheses on subsequent levels.

It is easy to see that in the general case in order to prove theorem from the hypotheses some additional assumptions might be needed about the interrelations of the primitive concepts. These will be called *verification conditions* and they will be treated as lemmas for the given level. In this

way in order to carry out the verification of all levels it is necessary to generate /and prove/ the - preferably minimal - verification conditions for each level and to handle the "inter-level" references of the primitives with the help of a *cross-references list of primitives* defined and used on the various levels. Verification by hand is hard; the program VERGEN /see [15], [22] and [23]/ is the first step toward automatising this.

In VERGEN procedure definitions are given in VDL, but the language of the specifications, /i.e. pre- and post-conditions for the procedures/ is not restricted. These can be arbitrary texts /in accordance with the requirements of interactive program design/; the important thing is that they describe the requirements of the primitives /black-boxes/ with a *precision that corresponds to the given SAM-level*. In order to generate the verification conditions we must give together with the VDL algorithms an appropriate system of axioms and rules of inferences /see [7] and [22]/. The VERGEN program accepts a *two-component* /algorithm description and requirement description or specification/ *language*. During the processing of the algorithmic definitions and the corresponding specifications the system generates verification conditions for the procedures /using a simple parameter correspondence scheme, see [Good, 70]/. Assuming the trivial conditions proven, it prints the others in a nice format "courteously" leaving room on the paper for the proof /to be done - at present - by hand/.

In *later versions of VERGEN*, taking into account the user requirements the following problems must be solved:

- definition of an algorithm description language more suitable for *design purposes*,
- more aspects /e.g. typechecking of parameters in the case of procedure-calls/ should be considered during verification condition generation,
- development of a theorem-prover mechanism, which the system can use to *prove* the non-trivial verification conditions genrated by the system itself.

2.3 Program design

We described the process of SAM-preparation, showing that the *SAM-like problem elaboration can be a method of the con-construction /in a structured manner/ of provably correct, well-structured program designs*. Thus we have a method of program design; a description prepared by using this method can be easily and unambiguously read and understood.

A *Software Support System* can be developed to assist program design by this method. The core of such a system can be the above mentioned VERGEN program. Some other important features of the system might be the following:

1. the above mentioned *verification facility* of the system should be *modifiable;* the user should be able to give a "knowledge" base /in the form of axioms and deduction rules/ which can be used by the system to prove more complex verification conditions;
2. implementation of a *query subsystem* /described in [16]/ using as a data-base a SAM-description that is tree-structured according to the members of some program family /Software Encyclopedia, see section 3.1/. Using this and the answers given by the designer for its questions the system can traverse an appropriate path in the tree while generating in a well-documented manner the program family member requested by the user;
3. provision of an *environment* which can be used to examine the *behaviour, usefulness, optimality* /in a given sense/, etc. of a SAM-description of any level using an appropriate */abstract/ test-bed* generated from the specifications.

In the definition of the features of a possible SAM Support System we must keep in mind the basic requirement that a system like this /i.e. one that is to be used as an aid in top-down, structured, verifying program elaboration/ should *communicate* with the user - at "design time" - *at several levels*.

A system like this - in view of the above - is envisaged as being built around some /abstract/ language or machine at the bottom level; assuming that *its* "abstract operations" have already been proven correct.

Thus *the task of the designer/implementor* may now be defined as one having to refine the problem definition /the

primary or original version/ using the above described means until the bottom level of the refinement process is the bottom level defined above, or a higher level which is algorithmically known to the system and is proven correct. /Note that this base-language can be viewed as the "Machine Oriented Language" /MOL/ of an abstract machine./ This Support System will be based on some sort of *deduction mechanism* to be used during generating conditions. This system can be built in such a way that it asks the user /who is in *interactive communication* with it/ to *prove the verification conditions* generated by it at the various levels of the description under examination. On the other hand from the automated aspect of such a system we would expect that it uses a *theorem-proving* subsystem to prove the verification conditions, and it should only turn to the user when it is in trouble.

2.4 Model implementation questions

So far we have shown the advantages of SAM-aided program design. The previously mentioned Support System will help in the implementation problems as well, and may perform such additional tasks as the generation of test-beds for given run-time environments.

The *implementation* of the abstract algorithms described /currently/ in VDL would belong to the tasks of this Support System. Using VDL as the language providing the abstract description formalism there are the usual two ways to implementation: interpretation, and translation to an implemented object language. In the former case we have immediate storage bounds problems. Translation of VDL is not an easy task either, since it is difficult, to find a usable /abstract/ object language that is implemented. Bridging the gap between the abstract description and concrete data representation is also problematic. We *experimented with using CDL* for implementation* purposes; CDL has a control structure that is

---

* Compiler Description Language [Koster, 71]

similar to that of VDL. CDL versions of VDL algorithms can be given relatively easily. Separate pre- and post-processors had to be used to resolve the differences between the abstract and concrete syntax. The following method has been successfully used in writing compilers /VERGEN was also designed using this method/;

- the abstract compiler written in VDL was translated - almost mechanically - to CDL by hand, the required interface was provided; in parallel with this
- the difference in the abstract and concrete levels was resolved by doing the parsing and code-generation of the compiler in CDL /omitting VDL completely/. The papers [10], [11] and [12] give the VDL design of PASCAL and BCPL compilers; the latter summarizes the experiences of *implementing the VDL design in CDL;* CDL output listings are provided.

Up to now we seperated design and implementation. The final goal of both activities is the definition in some programming language of the /proven correct/ algorithm of the solution of the problem. The final solution of this problem would be the expansion of the research into the area *of automated problem solving.*

To summarize, the long-range goal of SAM-research is the creation of such an automatic problem solving system in which *programming is done by problem specification.* In the current phase of the research we concentrate our efforts to develop methods for giving "good" descriptions of SAM-s - and implementing them - keeping in mind the requirements of further development towards the direction of automated problem solving.

3. DESCRIPTION OF SOFTWARE ELEMENTS

In the previous section we discussed the methodological aspects of SAM-research; we shall now give an account of our results in the application area, in the formal description of software components.

First we shall outline our ideas about a "Sofware Encyclopedia"; we shall then examine the SAM-description of an abstract program production environment /this can be considered as a chapter of the Encyclopedia/. Results concerning description, design and implementation of compilers and other applications will also be given.

3.1 Software Encyclopedia

The purpose of SAM-research is to develop a design and implementation methodology which allows us to prepare hierarchically ordered, general, abstract, verified models of problem families. The Software Encyclopedia can be viewed as a *tree-structured graph of descriptions* that correspond to forms filled in as described in section 2; the nodes are these descriptions and the branches are the possible solutions of the problems described in the node they originate from. Thus the Encyclopedia describing a problem family can be viewed as an actual "*family-tree*", which is

- a description in which by choosing /by appropriate decisions/ among the alternatives at the various levels a path can be traversed in the tree; in other words the Encyclopedia contains its own directions for use,
- starting from the first level, if during the above steps we stop on some level, then the level-descriptions on the traversed path give the description of an element of the program family. This is a description /or program/ that uses primitives which have remained undefined down to this level. Now if there is an *abstract machine* which "understands" these

primitives, then we have an *implemented* version of the chosen family member.

It should be emphasized that these are only ideas. It is obvious that the use of a contemplated Encyclopedia - as a handbook - has many advantages; in the construction of well-documented, well-structured and proven correct program designs as well as in evaluating the finished software product and in teaching about software elements. This may make the Encyclopedia useful for the designer and customer of the software product, as well as for the educator teaching computer science. For a detailed discussion of these application possibilities see [5].

The development of the descriptional tools of SAM-s will happen in parallel with that of our long-range goal, the Problem Solving System. Using the current set of tools /e.g. VDL/ the experimental realization of chapters of the Software Encyclopedia is currently in process at our institute. Results to date are described in [Dömölki, 73], [7], [8], [9], [16], [17], [20] and [21]. These give SAM-descriptions of elements of the program production environment /e.g. assemblers/ and other software elements; they will be summarized later in this section.

3.2 Description of the elements of the program-production environment

An /imaginary/ Software Encyclopedia describing a *small-computer environment* used for traditional functions might be, for the purposes of this paper, divided into three "volumes":

a/ the components interpreting or compiling higher-level languages,

b/ the elements of a so-called program production environment responsible for the conversion of some programs written on a /macro/ assembly level language to other programs that can be run by the operating system,

c/ other software elements /the operating system, its components such as a file management system, etc./.

The relation to machine-dependence of the above volumes is not the same. The high-level languages - in volume a/ - are usually designed with machine-independence in mind. The application of the SAM-method is more interesting in the case of the other two volumes since presenting the *common, general, implementation and machine-independent features of the elements* of these volumes may help to solve many problems /e.g. portability/.

With respect to volume b/ a survey of assemblers macro assemblers and editors has been made in [2], together with a VDL description of the corresponding software components of some concrete machines (including IBM 360/370). The following general - structured - models have been prepared so far: the macro assembler /macro processing and assembly treated separately/, the linkage editor, the loader and a tracing system.

The first paper to be mentioned in this connection and quoted already, [Dömölki, 73] gives an *Abstract Assembly Model*. This is a SAM-description of a general assembler /i.e. the compiler semantics of a general assembler language/. Of special interest is that this model shows the machine- and implementation-independent aspects of /otherwise very machine-dependent/ assemblers, thus elucidating the essence of these programs /see also [Varga, 76a]/.

The paper [9] gives a parallel description of a one- and a two-pass assembler. This paper gives a more implementation-dependent version of the Abstract Assembler Model introduced in the previous paper /that is it can be used in an actual program design/. It also shows that it is possible to give SAM that describes the various *phases of assembler-level program production* at once /i.e. assembly, editing, loading/; that is description of a program family can be given introducing the phases as alternatives defined by appropriate possibilities.

Table 2. gives the description of several levels of the assembler-SAM discussed in [Dömölki, 73] and [9], together with an exposition of the problem to be solved and the corresponding decision made at each level.

The *Macro Assembler Model* described in [8] gives the macro additions required for the two previous assembler models. This paper, besides emphasizing the general and common characteristics of small-computer macro assemblers, contains a good description of all the features of the IBM 360/370 macro assemblers in a suitably generalized form.

While the previous SAM is the description of a single, general macro processor, [16] contains a description of a whole family of /small-computer/ *macro-assemblers, a tree-structured SAM.* The members of the family are not introduced with the method of parallel levels seen in [9]. As the alternatives appear due to the design decisions, they live their independent existence as branches of the decision tree. A common feature they have - apart from the common ancestry - is that the problems /determining the characteristics of the levels which appear/ obviously some will appear only with some alternatives. There are four libraries, in the tree-structured macro assembler SAM-description /the problems and possible solutions, the VDL-instructions, the syntactic rules and the primitives/. The designer references these libraries like a macro-call in the various sections of the SAM-form; this saves a lot of repetition. The specifications of the procedure-primitives are written in a form acceptable to the VERGEN program /see 2.2/. This "chapter" of the Software Encyclopedia /dealing with macro assemblers/ shows five levels of the family-tree describing about 64 alternatives. Thus an interesting experiment is described in this paper containing important lessons for the future editors of the Software Encyclopedia in connection with the enjoyable, readable /that is with computer supported/ SAM-descriptions to be contained therein. Table 3. illustrates the first pages of the "problems and possible solutions" library of [16].

Table 2. First few levels of a SAM for assemblers (problem, decision)

LEVEL	*PROBLEM*	*DECISION*
A	Basic structure of the assembly module and the assembler	Assembly module consists of declarations and program part. Program is processed first, declarations - containing information about external and entry names only - afterwards.
B	Processing of the program part	Program-part is a *list* of statements, to be processed in a serial order.
C	Types of statement and their processing	Three types of statements are used, in all three an *expression* is computed and the obtained *value* is either (1) assigned to a name (assignment), or (2) given to the location-counter (1c *modification*), or (3) used to fill machine-words of the output (*content-definition*). In the last case the value is *adjusted* to the previously given *length*, the result is inserted to the output component BODY.
D	Initialization of expression evaluation	Before the actual computation of an expression takes place, it should be *checked* whether all information needed for the computation is available or not. This check means a pre-processing of the expression and its result is used both by the actual calculation of the expression or by the composition of an *undefined-indication*, containing all information for the postponed computation of the expression, when it becomes defined.
E-H	Handling of - possibly postdefined - names	Values assigned to names are stored in an dictionary. A name may occur is in expression before a value is assigned to that name and this can be the reason of the expression being undefined. In such cases the *undefined-indications*, obtained as the result of the computation of the expression, are stored instead of the corresponding values and *references* are set up in the dictionary to point from the undefined names to the corresponding undefined-indications. When a value is assigned to a name, these references are resolved.

LEVEL	*PROBLEM*	*DECISION*
I	Calculation of the value of expressions	*Expressions* are constructed from names, constants and location-counter values by infix *operators*. Calculation of the expression is defined recursively.
.....	.....	.....
N	Structure of the dictionary	Dictionary is a set of dictionary-items selected by *names*. Each item has value and reference components.
0	Structure of the output component (BODY)	BODY is a *list* of body-values; i.e. *values* (addresses, consisting of a *base* and displacement, or numerical values), lc-directives or undefined-contents.

Table 3. First pages of the "problem and possible solutions" library of SAM-tree for macro assembler

NUMB.	*PROBLEM*	*POSSIBILITIES*
1.	In which phases of program production is it useful to apply the textual substitution provided by macro facilites?	1. Before lexical analysis 2. During syntax analysis 3. During object code generation 4. During linking 5. At load time
2.	What is the (assembly level) syntactic unit which will be produced after the text substitution?	1. One or more assembly lines which represent a higher level syntactic unit (e.g. declaration part) 2. A block which can be empty or can contain one or more assembly lines; in the latter case these form a syntactic unit 3. A component of a single assembly line (e.g. label)
3.	Determination of the relationship of macrogenerator and assembler	1. The macrogenerator knows the history of the assembly to this point; during substitution it can use knowledge about the low level syntactic units of the assembler language (e.g. attributes of identifiers), it has access to the assembler's tables 2. The macrogenerator has only limited knowledge about the syntax of the assembler language i.e. that it consists of lines and so the expander itself has to generate lines. The macro operations and the assembly are separable both logically and in time
4.	Definition of the basic syntactic character of the source text	1. The source text is a list of records 2. The source text is a list of characters
5.	Besides explicit macro-calls are implicit macro-calls to be allowed?	1. Yes; macro-calls are generated during the processing of the source text based on the built in knowledge of the macroassembler 2. Only explicit and positionally fixed macro-calls are allowed 3. Explicit but positionally independent (i.e. condition dependent) macro-calls are allowed

NUMB.	*PROBLEM*	*POSSIBILITIES*
		4. A combination of 1. and 3.: some (e.g. standard) macros are expanded according to general rules, others are positional
6.	The syntax of macro-calls is fixed or it can change	1. Fixed 2. It can change (see extensible languages)
7.	The character of the syntax of macro-calls	1. Explicit 2. Implicit 3. Combination of 1. and 2.
8.	Does the assembler or the macrogenerator have priority in the analysis of the higher level syntactic units of the source text?	1. Assembler has priority 2. Macrogenerator has priority 3. Strategies 1. and 2. can be switched according to text context or special directives

The paper [17] contains the SAM-description that is so far the most "readable"; it is the detailed elaboration of a single alternative of a *general program-tracing system*, such that the textual and the VDL-based algorithm descriptions are readable and understandable on a standalone basis; as well as being nicely complementary and explanatory of each other when read together.

Table 4. is a brief summary of the problem and decision sections of the informal part of the first levels of [17]. In the model levels D-J introduce the /user/ commands used to initiate the required trace; we only give the refinement of the "trap handling" commands introduced on level E.

There are several other papers in preparation in this area. We refer here to [21] /under publication/ which *describes a general structured abstract model of the program production environment.*

3.3 Description, design and implementation of compilers

This section reports on our results concerning the formal description of higher-level languages and their compilers. As mentioned before, our starting point in the formal description of SAM-s was VDL, which was originally designed for the formal description of semantics programming languages. The first practical applications for the definition of the abstract semantics of PL/I, ALGOL 60 and BASIC are well known. At our institute we first used VDL to give the interpretive semantics of APL, see [1]. Of special interest in this paper is the fact that it emphasizes the interactive features of an APL system, our first effort of this kind of application.

The paper [6] in the *description of the compiler of a very simple language*, based on [McCarthy, 67], where *the design is proven correct*. The notation of the abstract compiler is defined; it is a VDL abstract machine which translates the objects satisfying the abstract syntax of the source language

Fig.4 *First few levels of a SAM for tracing system (problem, decision)*

LEVEL	*PROBLEM*	*DECISION*
A	Definition of the basic structure of the program to be traced; definition of the main steps of tracing.	The two input components of tracing are: the *program* to be traced and the *commands* specifying the (kind of) trace. The trace consists of an *initialization* activity and the execution of the program *one instruction at a time*. Some of the instructions of the program are so called *trap* instructions; the execution of one of these is interpreted as the insertion of a *tracing step*. Other instructions are left undefined for the purposes of this model.
B	Definition of the format of user commands.	The commands form *command-gropus*. Both the initialization activity and the tracing step mean the execution of given command-groups.
C	What is the structure of a command-group? How should a command-group be interpreted?	A command-group is a *list of commands;* one of these must be a special "return" command. The execution of a command-group means the execution of the individual commands *in sequence until a return command is reached*. The interpretation of an individual command should consist of the execution of some sort of tracing activity and the selection of the next command to be interpreted.
D	What kinds of commands do we need? How should these be interpreted?	A command *requesting information* about the current status of the running program, *trap-handling* commands, *control-sequencing* commands which allow the modification of the order of execution of the commands, an *inquiry* command which allows the examination of the current program status, an *end* command specifying program termination, a *newcommand* command which allows modification of commands "on the fly" are allowed in the model.
E	Definition of the interpretation of the traphandling commands.	The trap-handler commands can be *trap-establishment* or *trap-removal* commands; these will specify a program address (using some sort of *address definition*) and a *command-group*. The trap-establishment command is interpreted as *placing a trap at the given address* and *establishing a correspondence between the address and the command-group;* the trap-

LEVEL	*PROBLEM*	*DECISION*
		removal command is interpreted as the *destruction of the above correspondence* and the *removal of the trap* (if necessary).
F-J	⋮	⋮
K	What do we mean by trap-establishment and trap-removal?	*Trap-establishment* means that the instruction at the given program address is exchanged for a trap instruction, provided that a trap was not previously placed here, and the original instruction at the address is recorded. A *trap is removed* if all command-group correspondences with this address are desolved; in this case the original instruction is replaced at the given address.
L	How is a correspondence established between an address and a command-group and how is such a correspondence desolved?	When a *correspondence is established between a command-group and a given address* the command-group is recorded with respect to the given address in such a way that a given command-group's correspondence to a given address be maintained uniquely even after several requests for the establishment of the same correspondence. Now the *removal of the correspondence* can be achieved by the removal of the single record of the given relation.
M	What is meant by the *address definition* mentioned on level E?	The *address definition* given in trap-handling commands can be an *address* or an address reference which in a given state of tracing defines an address; of these the model allows for the use of the *address of the next instruction to be executed*, the *current address*, the *start address* of the program and in case of subroutine calls the *return address*.

into objects satisfying that of the target language. We can say it is the plan of the concrete compiler and it can deal with the semantics of the source language without taking into account details of syntax. Assuming that the interpretive /abstract/ semantics of the source and object language are given, the correctness of the compiler written in VDL can be proved by showing the equivalence of the interpretive and compiler semantics of the two languages. This is illustrated in Fig.3.

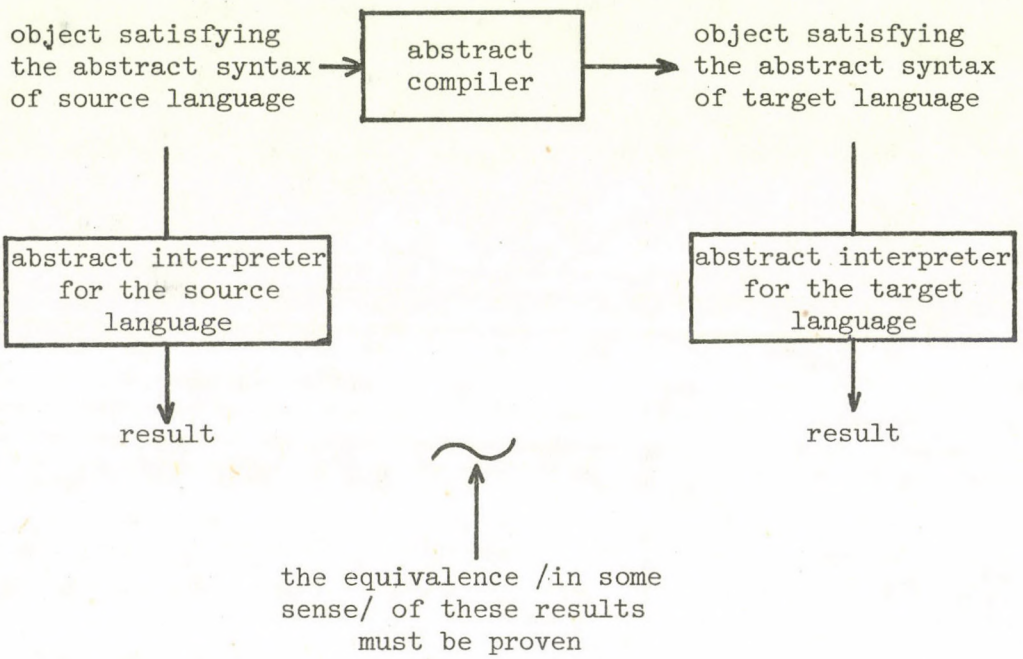


Fig.3: Equivalence of the interpretive- and compiler-semantics of languages

The abstract compiler mentioned above constitutes the core of a compiler construction method. According to this method the production process consists of two phases. The first phase separates into three independent activities:

i/ implementation of the lexical analyzer /scanner/
ii/ implementation of syntax analyzer /parser/
iii/ definition of the abstract compiler.

Since these activities are essentially independent they can be carried out and verified in parallel. You can verify formally (e.g. in the case of iii/ as described earlier), informally or by testing (e.g. in the case of i/ or ii/ if you have no better tools).

In the second phase of the construction process the "only" task is to put together the scanner and the parser and to "decorate" it with semantic actions which are a concrete realization of the abstract compiler. The places of the insertions are presented by the abstract compiler as well. It was found that using VDL as a definition language for the abstract compiler and CDL as an implementation language makes this process quite mechanical. Since the elements to be linked together are already proved to be correct, it is much easier to verify the whole concrete compiler.

The method described above has been used in several projects. A two-pass *BCPL compiler* was written. The experiments of this method in this project are analysed in [12]. The abstract *compiler for PASCAL* in shown in [10] and [11]. A BASIC interpreter is under development using [Lee, 72a]' definition of BASIC in VDL. In [18] we shall try to construct a new *description for BASIC* using the new definition method proposed by the Vienna Laboratories in 1974 /see [Bekic, 72]/. In all these projects the design is in VDL, the implementation in CDL.

3.4 Description of other programs

Of other applications we mention a description of the FIND program, introduced in [Hoare, 71b]. In [7] a structured VDL version of this program is used to illustrate the axioms and inference rules introduced for VDL in the same paper. A summary of the definitions and specifications of the VDL procedures for FIND is given in Table 5. /where □ stands for PASS, ~ stands for "is a permutation of" and the variables p and q are always bounded by a universal quantifier. Some procedures have two

PRE	DEF	POST
$1 \leq f \leq \text{length}(A)$	find(A,f)=reduce(A,f,1,length(A))	$(1 \leq p < f \leq q \leq \text{length}(A) \supset \square_p < \square_f < \square_q) \wedge \square \sim A$
$P(A,f,m,n) \overset{d}{\equiv}$ $(1 \leq p < m \leq r \leq n < q \leq \text{length}(A) \supset A_p < A_r < A_q)$ $\wedge m \leq f \leq n$	reduce(A,f,m,n)= m<n → reduce(vec(x),f,lb(x), ub(x)); x:order(A,f,m,n) T → PASS:A	$P(\square,f,f,f) \wedge \square \sim A$
P(A,f,m,n)	order(A,f,m,n)=ord(A,f,m,m,n,n,A_f)	$P(\text{vec}(\square),f,\text{lb}(\square),\text{ub}(\square)) \wedge \text{vec}(\square) \sim A$
$R(A,f,m,g,h,n,s) \overset{d}{\equiv}$ $(m \leq p < g \supset A_p < s)$ $\wedge (h < q \leq n \supset s < A_q)$ $\wedge P(A,f,m,n)$	ord(A,f,m,g,h,n,s)= continue(B,f,m,i,j,n,s); B:change(A,i,j); i:up(A,g,s), j:down(A,h,s)	$P(\text{vec}(\square),f,\text{lb}(\square),\text{ub}(\square)) \wedge \text{vec}(\square) \sim A$
$(i \leq j \supset R(B,f,m,i+1,j-1,n,s))$ $\wedge (j < i \supset R(B,f,m,i,j,n,s))$	continue(B,f,m,i,j,n,s)= i≤j → ord(B,f,m,i+1,j-1,n,s) i≤f → PASS:μ_o(<vec:B>,<lb:i>, <ub:n>) f≤j → PASS:μ_o(<vec:B>,<lb:m>, <ub:j>) T → PASS:μ_o(<vec:B>,<lb:f>, <ub:f>)	$P(\text{vec}(\square),f,\text{lb}(\square),\text{ub}(\square)) \wedge \text{vec}(\square) \sim B$
$R(A,f,m,i,j,n,s) \wedge A_j \leq s \leq A_i$	change(A,i,j)= PASS: μ(A;<elem(i):elem(j,A)>, <elem(j):elem(i,A)>)	$(i \leq j \supset R(\square,f,m,i+1,j-1,n,s)) \wedge$ $(j < i \supset R(\square,f,m,i,j,n,s))$
$1 \leq i,j \leq \text{length}(A)$		$\square_i = A_j \wedge \square_j = A_i \wedge (p \neq i \wedge p \neq j \supset \square p = A_p)$
R(A,f,m,g,h,n,s)	up(A,g,s)= A_g<s → up(A,g+1,s) T → PASS:g	$R(A,f,m,\square,h,n,s) \wedge s \leq A_\square$
$m \leq p < g \supset A_p < s$		$(m \leq p < \square \supset A_p < s) \wedge s \leq A_\square$
R(A,f,m,g,h,n,s)	down(A,h,s)= s<A_h → down(A,h-1,s) T → PASS:h	$R(A,f,m,g,\square,n,s) \wedge A_\square \leq s$
$h < p \leq n \supset s \leq A_p$		$(\square < p \leq n \supset s \leq A_p) \wedge A_\square \leq s$

Table 5. VDL definitions and procedure specifications for FIND

pairs of PRE- and POST-conditions as specification, in such cases the upper one is the assumption used at the place where the procedure is called, while the lower one is a theorem, which can be proved from the definition, and implies the assumption/.

The verification conditions for this description generated by VERGEN can be found in [15]. This paper describes *a few levels of VERGEN* itself illustrated by the listing generated by VERGEN for these levels.

The [20] paper gives a possible *SAM-description of a general small-computer file management system*. A brief summary of the problems and definitions section of the first levels of this model is given is Table 6.

We are also planning the preparation of the SAM-s of *several other operating system components*. This is partly to satisfy the experimental requirements of the methodology research, partly to continue with the development of the Software Encyclopedia itself.

Table 6. First few levels of a SAM for file-management system (problem, decision)

LEVEL	*PROBLEM*	*DECISION*
α	The definition of the basic structure of the system executing the user programs and the definition of the basic structure of the program to be executed.	Some of the program's instructions are special *file-handling instructions*. The system executes the program instruction by instruction until a stop instruction is reached. The model will only consider the file-handling instructions.
A	Definition of the essential structure of files; the main steps of grouping and interpreting file-handling commands.	A file consists of two parts: the *header* which contains information about the file as an organized unit of data, and the *body* which contains the user data proper. The file-handling instructions are of two kinds: *preparation* /administration/ instructions and *data-handling* instructions which operate on the header and the body respectively. The interpretation of the file-handling commands consists of a /security/ *validation* and depending on the result of this *execution of the required action* or the *generation of an error report*.
B	What is the validation condition required for the interpretation of file-handling instructions?	Every file has a corresponding *file description table* which the system uses to record the current status of the file during processing. The table contains an *opening flag* which indicates that a given file at a given time is ready or not for processing. Examination of this flag is the validation step. Data-handling instruction may only be executed when the file is open; the preparation instructions OPEN only when the file is closed, the instruction CLOSE only when the file is open.
C	Definition of the structure of the body of the file and of the unit of data accessible by the data-handling instructions.	The file consists of *records* /logically connected units which are moved together/. The data-handling instructions manipulating records. These consist of a *secondary validation*, the *required manipulation* or the generation of an *error report*. There are four types of data-handling instructions /READ, WRITE, REWRITE, DELETE/. The

LEVEL	*PROBLEM*	*DECISION*
		secondary validation checks whether the required operation can be performed at the given time.
D	The main stops of performing the individual record operations.	The record operations are performed by the system in three main steps: *it determines the position of the required record it checks the record,* and depending on the result of the check it performs the required *transput* or transfers control to a predetermined continuation address. The condition of the transput in the case of the WRITE instruction is that the required record be *not* in the file, in the other cases that it should be there.
E	How is the transput of the record actually performed?	Within the file the records form blocks, these are the units of physical data transmission. During the transput of a record the block containing the record is transmitted first, if required /this is performed by physical file-handling routines/, the actual operation is then performed on the record as it resides within the block.
F . . I	How is the file constructed from records, how are the records handled?	The model provides for three types of file organization; sequential, relative and indexed-sequential. Two types of file-access are treated: sequential and random. In the several different combinations the *record position* is determined in a different manner and some of the administrative actions are performed differently.
J	Determination of the main tasks of the preparation instructions.	The OPEN instruction provides permission to process the file in the manner supplied by the *opening mode* /input, output or update/. During the interpretation of the instruction the system checks whether the opening mode is compatible with the information in the file description table and the file header; if so the opening flag is set "FALSE", thus no other processing can be performed on the file until the next OPEN instruction.
K	What are the secondary validation conditions of the interpretation of the data-handling instructions?	The secondary validation applies to whether the *operation type* of the instruction and the access mode is compatible with the opening mode and the file organization

4. APPLICATION POSSIBILITIES

The paper has presented a set of formal tools for specification, design and implementation of software objects.

The method of Structured Abstract Models enables the programmer to describe the general and abstract features of programs and to develop a whole family of programs in a top-down and verified manner. The data structures and algorithms are to be presented in an abstract program specification language.

Using the *method* of Structured Abstract Models - restricting our objects to programs, software elements - it is possible

- to work out a *design and implementation methodology* which in compliance with the rules of top-down, structured problem solving, is based on the determination of the hierarchical order of decisions, and allows *verification* to be carried out in parallel with this;
- to give a *formal description of software products* which can show the appropriate concrete /or perhaps only hypothetical, unimplemented/ software products ordered by the decision hierarchy defined by the user's order of priorities.

Finally let us review in which phases of software production we may use our models of software components:

- in the *evaluation* of a given product /to help customers to *choose* from several alternatives/;
- in the *specifications* /problem definition/ of a new product;
- in the preparation of a *verifiably correct design plan*, and
- during the *implementation of a well-documented, error-free product*.

We should also mention the advantages of a clear, well-

structured description /i.e. SAM/ of the functions of the computer operator as an example of a non-software product application of SAM-s.

The *educational* importance of the method must also be noted; the possibilities to use models of problem families /"Software Encyclopedia"/ in teaching should be exploited. We should also note that our universities in recent years have in fact started to utilize this possibility; [Varga, 76a-b] are good examples of this.

ACKNOWLEDGEMENT

The research reported in this paper was supported by the Hungarian National Bureau for Computer Applications. The authors gratefully acknowledge the help they received from Zs.Farkas and T.Langer in writing some parts of the paper. They, together with J.Aszalós and I.Siklósi formed the core of the team engaged in this research. Valuable help was received from P.Köves in preparing the English text.

REFERENCES

[Bekic, 74] H.Bekic, D.Bjorner, W.Henhapl, C.B.Jones, P.Lucas:
A Formal Definition of a PL/I Subset
IBM Lab. Vienna, 1974. TR. 25.139.

[Boehm, 73] B.W.Boehm:
Software and Its Impact:
a Quantitative Assessment
Datamation, May, 1973. pp. 48-59.

[Chang, 73] Chin-Liang Chang, Richard Char-Ting Lee:
Symbolic Logic and Mechanical Theorem Proving
Academic Press, 1973. New-York

[Dijkstra, 68] E.W.Dijkstra:
The Structure of T.H.E. Multiprogramming System
Comm.ACM. Vol.11, No.5, May, 1968. pp. 341-346.

[Dijkstra, 70] E.W.Dijkstra:
Structured Programming
Software engineering techniques,
J.N.Buxton and Randell /Eds/.
NATO Scientific Affairs Division,
Brussels, Belgium 1970, pp.84-88.

[Dijkstra, 72] E.W.Dijkstra:
Notes on Structured Programming
APIC Studies in Data Processing No.8,
Academic Press, 1972. pp. 1-82.

[Dömölki, 73] B.Dömölki:
On the Formal Definition of Assembly Languages
Symposium and Summer School on the Mathematical Foundations of Computer Science
High Tatras, Czechoslovakia,
Sept, 1973. pp. 27-39.

[Good, 70] D.Good:
Toward a Man-Machine System for Proving Program Correctness
The Univ. of Wisconsin, Philadelphia,
1970. 70-72, 053. /Ph.D. thesis/

[Hoare, 69] C.A.R.Hoare:
An Axiomatic Basis of Computer Programming
ACM Vol.12, No.10, October, 1969.
pp.576-583.

[Hoare, 71a] C.A.R.Hoare:
Procedures and Parameters: an Axiomatic Approach
Symposium on the Semantics of Algorithmic Languages
Berlin-Heidelberg-New-York
Springer, 1971, pp. 101-117.

[Hoare, 71b] C.A.R.Hoare:
Proof of a Program: FIND
ACM, Vol.15, No.1, January, 1971.
pp.39-45.

[Hoare, 72a] C.A.R.Hoare, M.Clint:
Program Proving: Jumps and Functions
Acta Informatica, 1972.1, pp. 214-224.

[Hoare, 72b] C.A.R.Hoare:
Proof of Correctness of Data Representation
Acta Informatica, 1972.1, pp.271-281.

[Hoare, 72c] C.A.R.Hoare:
Prospects for a Better Programming Language
INFOTECH State of the Art Lectures on Programming Languages,
1972. London, pp. 327-343.

[Koster, 71] C.H.A.Koster:
A Compiler Compiler
MR 127/71, Mathematics Centrum,
Amsterdam.

[Lucas, 68] Lucas P., Laure P., Stigleitner H.:
Method and Notation for the Formal Definition of Programming Languages
IBM Lab.Vienna, 1968. TR 25087.

[Lee, 72a] John A.N.Lee:
The Formal Definition of the BASIC Language
The Computer Journal, Vol.15. No.1,
pp. 37-41.

[Lee, 72b] John A.N.Lee:
Computer Semantics
Van Nostrand Reinhold Co., 1972.

[McCarthy, 67] McCarthy J., Painter J.A.:
Correctness of a Compiler for Arithmetic Expressions
Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science, pp. 33-41.
/ed.Schwartz J.T.-. Providence, Rhode Island: American Mathematical Society

[Mills, 72] Harlan, D.Mills:
Mathematical Foundations for Structured Programming
International Business Machines Corporation, Gaithersburg, Maryland, 1972.

[Neuhold, 71] E.J.Neuhold:
The Formal Description of Programming Languages
IBM System Journal 1971, 2.pp.86-112.

[Varga, 76a] L.Varga:
The Abstractions of Machine Dependent Program Forms
KFKI-76-11, Budapest, 1976.

[Varga, 76b] L.Varga:
The VDL Graph
KFKI-76-28, Budapest, 1976.

[Wegner, 72] P.Wegner:
The Vienna Definition Language
ACM. Computing Surveys, Vol.4, No.1, 1972.March, pp. 5-63.

APPENDIX, LIST OF INTERNAL PAPERS, 1973-76

/in Hungarian/

[1] T.Langer:
The Formal Description of APL by Using Vienna Definition Language
Budapest, 1973. Inf. 1080/72.

[2] E.Sánta:
Assembly Languages and Assemblers /Survey/
Budapest, 1973.Inf. 1101/73.

[3] B.Dömölki:
Structured Abstract Models /in English/
Budapest, 1973.

[4] B.Dömölki, E.Sánta:
Some Aspects of the Foundation of Computer Science
SAM-I. Introduction
Budapest, 1974. Inf. 1368/74

[5] B.Dömölki, E.Sánta:
Structured Abstract Models -SAM- and their Use
SAM-I. Chapter 1.
Budapest, 1974. Inf. 1368/74.

[6] T.Langer:
Structured Abstract Compiler as a Tool for Verified Compiler Planning
SAM-I. Chapter 2.
Budapest, 1974. Inf. 1368/74.

[7] I.Siklósi:
Verification of VDL Programs
SAM-I. Chapter 3.
Budapest, 1974. Inf. 1368/74.

[8] J.Aszalós:
Structured Abstract Macroassembler Model
SAM-II. Chapter 1.
Budapest, 1974. Inf. 1433/74.

[9] E.Sánta:
Structured Abstract Assembler Model
SAM-II. Chapter 2.
Budapest, 1974. Inf. 1433/74.

[10] S.Bárány:
Compilation of PASCAL Statements
Diploma Thesis, Budapest, 1975.Inf. 1477/75.

[11] É.Janni:
The Universal Compiler-Compilation of PASCAL Expression
Diploma Thesis, Budapest, 1975. Inf. 1476/75.

[12] T.Langer:
Lessons of a Methodological Experiment /A BCPL Compiler Planning in VDL and Implementing in CDL/
Budapest, 1975.
Inf. 1498/1975.

[13] J.Bánkfalvi:
Symbolic Logic and Automatic Theorem Proving
SAM-III. Volume 1. Chapter 1.
Budapest, 1975. Inf. 1525/75.

[14] I.Sain:
Model Theory and Automatic Theorem Proving
SAM-III.Volume 1. Chapter 2.
Budapest, 1975. Inf. 1525/75.

[15] I.Siklósi:
Description of Program Verification Condition Generator, VERGEN
SAM-III. Volume 2.
Budapest, 1975. Inf. 1564/75.

[16] J.Aszalós:
Family of Structured Abstract Models for Macroassemblers
SAM-III. Volume 3.
Budapest, 1976. Inf. 1555/75.

[17] Zs.Farkas:
Structured Abstract Model for Trace System
SAM-III. Volume 4.
Budapest, 1975. Inf. 1557/75.

[18] L.Verbovszki:
A New Method for the Description the Semantics of Programming Languages and its Application in the Case of BASIC
Diploma Thesis, Budapest, 1976. SZÀMKI 1592/76.

[19] J.Aszalós:
An Overview of Structured Programming Techniques
SAM-IV. Volume 2.
Budapest, 1976. /forthcoming/

[20] G.Szendi:
Structured Abstract Model for File Management System
SAM-IV. Volume 1., Budapest, 1976. SZÀMKI 1635/76.

[21] B.Dömölki, Zs.Farkas, E.Sánta:
Structured Abstract Models for Program
Production Environment
SAM-IV. Volume 3. /forthcoming/

[22] I.Siklósi:
Proving of Structured Abstract Programs /SAM-s/
Diploma Thesis, Budapest, 1976.SZÁMKI 1589/76.

[23] K.Krasnyánszki:
Methods of Program Proving
Verification of VDL Programs.
Diploma Thesis, Szeged, 1976.

CELLULAR DESIGN PRINCIPLES
A CASE STUDY OF MAXIMUM SELECTION
IN CODD-ICRA CELLULAR SPACE

Part One:

TOOLS PREPARATIONS

G.Fay

CSM HTG

Budapest, Hungary

CHAPTER 0.

SUMMARY

A cellular automaton has been designed in CODD-ICRA space for the selection of the maximal number out of a given set of positive integers. The maximum selecting cellular automaton is called MAXEL. It consists of $k \times \ell$ modules arranged by k rows and ℓ columns, where k is the number of bits of the ℓ numbers out of which the maximal one is to be selected.

A MAXEL module takes roughly 40x50 cells /strictly 34x49/, the whole MAXEL takes somewhat more than $40\ell \times 50k$ cells for a frame, of about 10 cells in width, necessary for the integration of the modules. So the size of MAXEL is approximately /depending on some eventual newer minor design tricks/:

$$(40\ell+20) \times (50k+20) \text{ cells.}$$

MAXEL's operation is two-staged. At the first stage the machine is set by the data to become a filter /a selective absorber/ permitting only the maximal number to go, at the second stage. At the end of the first stage MAXEL sends a signal that can trigger the peripherials to dump the data on the MAXEL.

The throughput is:

$$\frac{\ell \text{ number}}{200k \text{ shot}} = \frac{\ell \text{ record}}{200 \text{ shot}} = \frac{\ell \text{ byte}}{8x200 \text{ shot}}$$

$$\ell \text{ bit} = \frac{1}{8} \quad \text{byte} = \ell \text{ record and k record} = \ell \text{ number}$$

The mode of operation of MAXEL is parallel.

Data are loaded parallelwise into the device, ℓ numbers simultaneously. Both stages take /about/ look shots. By way of illustration we take the following /today already feasible/ data: -

size of a cell: 1 mm^2
Cycle time of a cell: 1 microsecond /= 1 Shot/
number of numbers: $\ell = 256$
number of bits: $k = 50$

Then:

- The *construction time* of the device, i.e. the time during which the complete device can be written into the cellular space, is: less than 15 seconds.
- MAXEL's *throughput* selection speed/: 160 Kbyte/sec = 5000 records/sec = 25000 number/sec
- The *size* of the necessary cellular space is 20 m^2

The design could have been augmented both in space and time by a few percents. Such a tight design, however, would paralyse the studies aimed at developing newer transition functions.

In the paper some new concepts have been introduced that seem to be suitable for relational data processing. There are

35 references, 27 tables and 25 figures.

ACKNOWLEDGEMENT

This work has been motivated by D.V.Takács's doctoral thesis [TAKÁCS, 1975] done during her fellowship in UER de Mathematiques, Logique Formelle et Informatique Sorbonne V., Université René Descartes, under the leadership to Professeur J.RIGUET. I am greately indebted to her and to Prof.RIGUET for his drawing my attention to the problem and to her helpfulness and kindness. Also, my thanks are due to the whole ICRA TEAM whose members are too numerous to be listed here. I am grateful to my ex-institute KGM ISzSzI where all the tehnical aids have been ensured concerning preparation of the manuscript.

CHAPTER 1.

INTRODUCTION

The relation between constructing and designing automata by automata is somewhat similar to that of a chisel and the sculptor. If we want to take seriously the term "constructing automata by automata" in the era of artificial intelligence then we have to carry out researches in design aids, accessories, equipments and techniques that can be acquired by automata themselves.

No doubt, self-reproducing automata can be constructed since von Neumann /1966/ and Codd /1968/ but their design is, of course, outside their range.

If we study the designs of cellular automata /the only candidates for self-designing automata/ we can easily realize that a considerable part of the associated treadmill work offers itself to be computerized. Tracing the signals along paths /we mean Codd's automata/, checking their escaping the

latches, avoiding collisions, unduly replications etc. are quite mechanical activities. However, they are not to be confused with computer simulation of cellular automata. Perhaps it is "computerized cellular automata design" what we are talking about. Of course, to design cellular automata by traditional computer is tolerable only at the beginning. At a later stage, one hopes, cellular automata can be taught not only to be constructed /i.e. to make a copy of/ but also to be designed by each other.

In this report I would like to display some of the mechanizable design aids that can be performed by cellular automata of the CODD-ICRA type. Design work takes quite a lot of data processing. Now it turned out, that the data - mostly concerned with the events taking place within the device to be designed - are arranging themselves into a relational form. Relational data processing technique has been invented by Codd /1970/ and /1971/. Readers familiar with this just have a look at the tables in our report /in chapters 9 and 10/ listing all the data crucial to judge the operation of a cellular device. It will be quite obvious then, that those data cry for the data processing language DSL ALPHA.

After all, cellular automata design is, in essence, the inference from given functions to structures to be constructed. And these inferences can get a great deal of help from a language so effective as DSL ALPHA.

This is the idea behind our report that is supposed to be a bit more than a mere design manual for a manual design. A number of new concepts have been introduced, concerning design, suggested for further refinements, and checked by the case study of a particular device design, a maximum selector.

In addition to this endeauvor for "automating cellular automata design" we believe that the concepts suitable for relational data processing will help the study of design correctness. In traditional computer science new theories are developing concerning program correctness proof. /See eg.Hoare, 1969/. There is - to my knowledge - no technique, whatever,

concerning design correctness proof of cellular machines. Simulation is, of course, no proof for correctness /or incorrectness/ because it does not tell reasons for phenomena.

Fortunately, there is already a bridge between Codd's cellular automata and Codd's relational data processing technique, due to Fay /1974/.

The paper is divided into two parts. The present first part is devoted to preparatory purposes so, here in the introduction, they can be avoided. The second part demonstrates the application of some recently introduced design concepts along the case study of the detailed design of a maximum selector. This whole paper intends to be quite technical, thus, at the outset we start with the problem and progressing from functions to structures and introduce the necessary concepts step by step. That's why cellular backgrounds are talked over as late as in chapter 4.

Ideas on design philosophy can be found again at the end of chapter 5.

The motivation of the present paper comes from D.V.Takacs's /1975/ doctoral thesis, dealing with the design of a cellular automaton CODD-ICRA cellular space for the first time.

CHAPTER 2.

FORMULATION OF THE TASK

2.1 The problem

Let us be given a set of ℓ numbers

$$N = (n^1, n^2, \ldots, n^j, \ldots, n^\ell)$$

each of which be given in a binary form of k bits:

$$n^j = \sum_{i=1}^{k} 2^{k-i} x_i^j = 2^{k-1}x_1 + 2^{k-2}x_2 + , \ldots , + 2^0 x_k .$$

The problem is to construct a cellular automaton in CODD-ICRA space which selects the maximal number/s/ out of the set. An algorithm, hopefully adaptable in CODD-ICRA, is as follows.

The basic idea behind the algorithm, to show below and which seems to be conveniently adaptable in CODD-ICRA space, is just the obvious fact that the maximal number is characterised by the most significant /leftmost/ bit except when the most significant bits of the numbers are all equal. In this case the next bit is relevant.

So it is at hand, as a first step, to resolute the set N into two /disjoint/ sets N^0 and N^1 containing all the numbers beginning with O and 1, respectively.

Let

1/ $$N = N^0 \cup N^1$$

where

2/ $$N^0 = \{n^j \mid [x_1^j = 0] \wedge [n^j \in N]\}$$

3/ $$N^1 = \{n^j \mid [x_1^j = 1] \wedge [n^j \in N]\}$$

Of course, one of N^0 and N^1 is non-empty. If N^1 is non-empty proceed the resolution with it, if empty take N^0. Thus one gets a tree where the leaf received by this way being a singleton contains the maximal number. Before elaborating the procedure, let's see an example.

Let

$$k = 5, \quad \ell = 8 \quad \text{and} \quad N$$

be given by the table below. /Table 2.1-1/

In this case

$$N^0 = \{n_1, n_3, n_5, n_7, n_8\},$$

$$N^1 = \{n_2, n_4, n_6\}.$$

Table 2.1-1 *Example for maximum selection*

j	n_j	x_1^j	x_2^j	x_3^j	x_4^j	x_5^j
1	4	0	0	1	0	0
2	25	1	1	0	0	1
3	13	0	1	1	0	1
4	27	1	1	0	1	1
5	2	0	0	0	1	0
6	18	1	0	0	1	0
7	14	0	1	1	1	0
8	6	0	0	1	1	0

with

	j	n^j	x_1^j	x_2^j	x_3^j	x_4^j	x_5^j
	1	4	0	0	1	0	0
	3	13	0	1	1	0	1
N^o:	5	2	0	0	0	1	0
	7	14	0	1	1	1	0
	8	6	0	0	1	1	0

and

	j	n^j	x_1^j	x_2^j	x_3^j	x_4^j	x_5^j
	2	25	1	1	0	0	1
N^1:	4	27	1	1	0	1	1
	6	18	1	0	0	1	0

As N^1 happens to be non-empty: we proceed with *its* resolution

$$N^1 = N^{10} \cup N^{11} = \{n^6\} \cup \{n^2, n^4\}$$

This time N^{11} is non-empty, so we can proceed with its resolution

$$N^{11} = N^{110} \cup N^{111} = \{n^2, n^4\} \cup \{-\}$$

This time N^{111} is empty, so we have to proceed with the other set, N^{110}, i.e.:

$$N^{110} = N^{1100} \cup N^{1101} = \{n^2\} \cup \{n^4\}$$

Here we received the set $N^{1101} = \{n^4\}$ being a singleton, containing the only element n^4, so

$$n^4 = 27$$

is the maximal member of set N.

This procedure is somewhat similar to that of the edge-notched card technique and although edge-notched card type selectors can be implemented in CODD ICRA, [Cf. FAY, 1974], we will see that our maximum selector differs radically from, and is much simpler than his edge-notched card selector.

On the other hand, the resolution technigue applied above is practically the same as the one frequently used in connection with Boolean funcitons, known as Shanon's expansion theorem.

In figure 2.1-1 we can see the associated tree of the procedure.

More formally, the procedure is, essentially, based on the formation of the sets of the form recursively defined by:

$$N^o = N,$$

$$N^{\varepsilon_1\varepsilon_2\ldots\varepsilon_{r+1}} = N^{(\varepsilon_1\varepsilon_2\varepsilon_3\ldots\varepsilon_r)\varepsilon_{r+1}} = \{n^j \mid [x^j_{r+1} = \varepsilon_{r+1}] \wedge [n^j \in N^{\varepsilon_1\varepsilon_2\ldots\varepsilon_r}]\}$$

for $\varepsilon_s = 0, 1;$ $s = 0, 1, 2, \ldots$

and producing the resolution,

$$N^{\varepsilon_1\varepsilon_2\ldots\varepsilon_{r+1}} = N^{(\varepsilon_1\varepsilon_2\ldots\varepsilon_r)0} \cup N^{(\varepsilon_1\varepsilon_2\ldots\varepsilon_r)1}$$

The maximal n^j will be contained in the first singleton whose last index is 1.

/In the example

$$N^{(\varepsilon_1\ldots\varepsilon_r)1} = N^{(\varepsilon_1\varepsilon_2\varepsilon_3)1} = N^{(110)1} = N^{1101} = \{n^4\}./$$

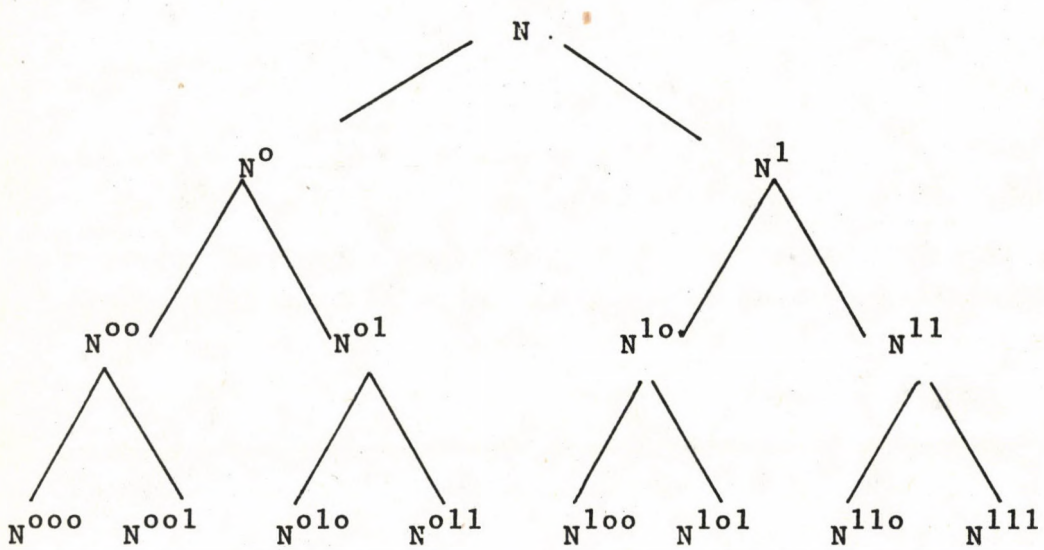


Figure 2.1-1

Binary tree associated with the maximum selection procedure

2.2 Reformulation of the problem

To get nearer to cellular aspects we give some reformulation of the problem approaching cellular term. To get hold of the maximal n's we have to examine all the bits x_i^j, for i = 1 ... k and j = 1,...,ℓ, to state whether $x_i^j = 0$ or $x_i^j = 1$ is the case.

If $x_i^j = 1$, then n^j is to be kept or included for, otherwise excluded from the next bit's examination. However, an essential exception is to be born in mind. We have to be convinced that no "allnought" case occurs. By an "allnought case", with respect to the i-th bit, it is meant that

$$x_i^1 = x_i^2 = x_i^3 = \ldots = x_i^\ell = 0.$$

In the allnought case one has to make a *correction* regarding the decision that n^j is to be excluded. In the allnought case n^j is still to be kept in spite of its leading bit x_i^j being zero.

Now these two queer operations "KEEP" and "EXCLUDE" however vague they seem to be, are very convenient to be implemented in CODD-ICRA cellular space. At this stage we refer to the block diagram in figure 2.2-1

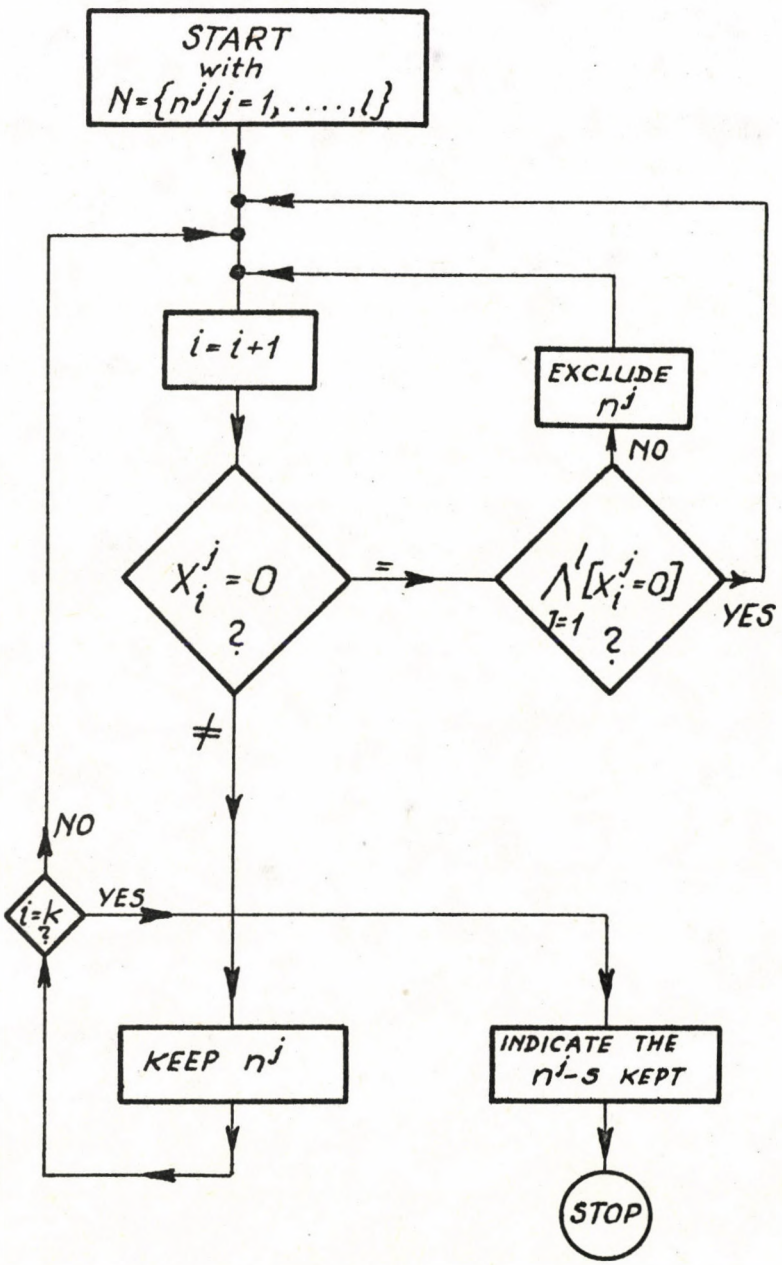


Figure 2.2-1

Block diagram for the reformulated algorithm

CHARAPTER 3.

BLACKBOX APPROACH

3.1 Basic operation principles of the device

Without any detailed knowledge of CODD-ICRA cellular space one can make some steps along system design. The only relevant knowledge is that is CODD-ICRA space Cancellation /"Exclusion from the examination"/ and Keeping a /representative of a/ number n^j is possible. In addition, of course, in CODD-ICRA the data can actually be transferred along paths. So, by figure 3.1-1 one can get the roughest idea about the principal functions of the device. Let the device to be designed, be called MAXELL /*Max*imum *Sel*ector/.

It is intuitively at hand to operate it by a two-stage mode of operation.

In the first stage one sets the inner gating mechanism performing the KEEP and EXCLUDE operations, then, in the second stage, one sends the data n^j through the paths controlled by the gates set during the first stage. As a result, all the n^j-s are killed /annihilited, cancelled, excluded/ except the maximal one/s/.

As it can be seen from figure 3.1-1, there are two systems of channels /paths in cellular terms/. The first is called the *information bus*, containing the paths labelled by

$$n_o^1,\ n_o^2,\ n_o^3,\ \ldots,\ n_o^j,\ \ldots,\ n_o^\ell$$

the *setting signals*, conveying the information /in the sense to be fixed later/ about the *data*

$$n^1,\ n^2,\ n^3,\ \ldots,\ n^j,\ \ldots,\ n^\ell$$

labelling the paths is the *data bus* in figure 3.1-1.

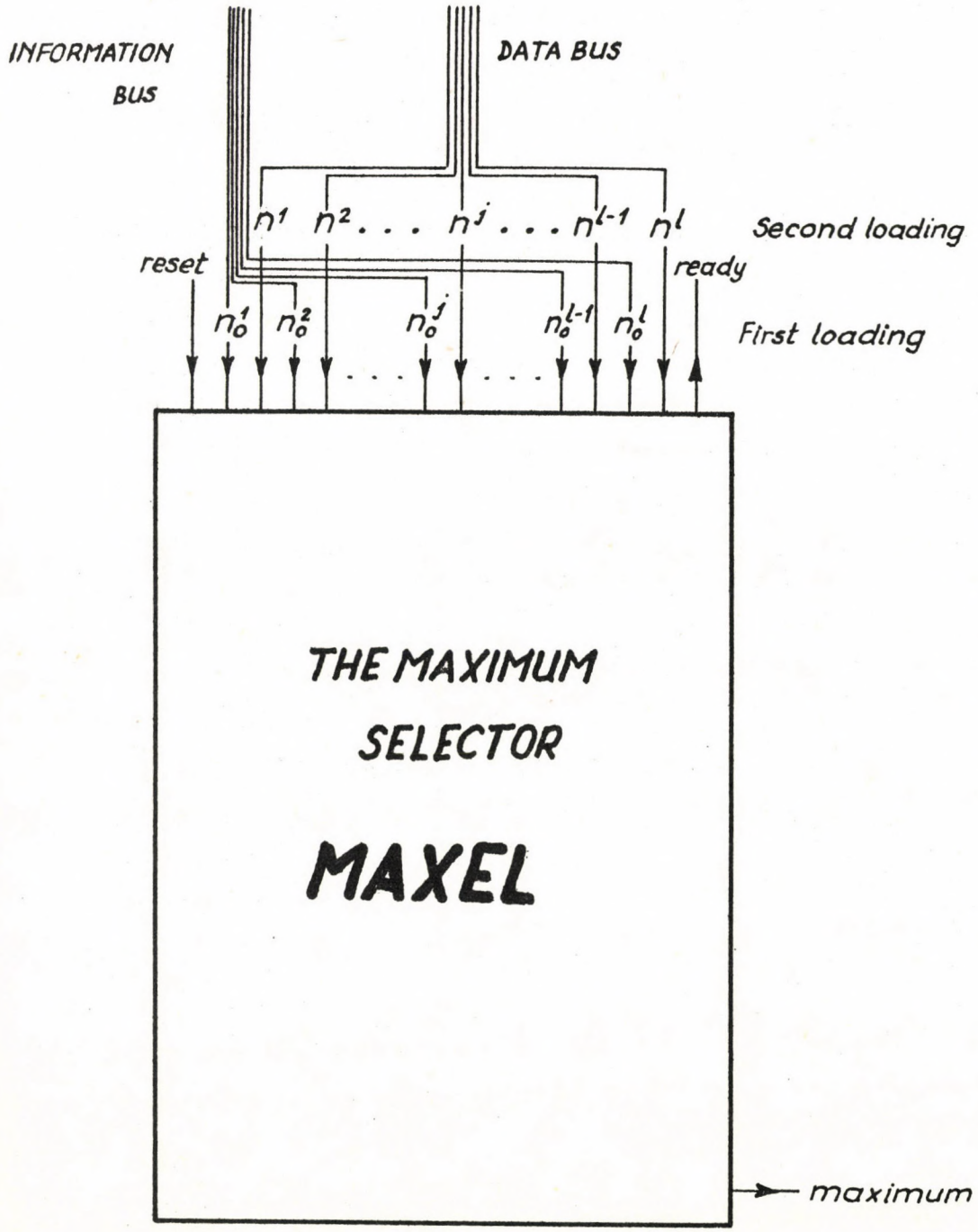


Figure 3.1-1

The basic principle of MAXEL'S two-staged operation

Having been properly set, MAXEL will, during the second stage, act as a *filter* /one should actually better say: "absorber"/ permitting only the maximum to cross it.

3.2 "Bread-dealing algorithm" and modularily

The logic of the "include" and "exclude" /see the block diagram on figure 2.2-1/, or "keep" and "exclude" /n^j/ shows a strong resemblance to that of the procedure practised in some college refractories when dealing out bread among the students using only one plate for the slices of bread. The plate travels /is passed/ from student to student with the instruction:

"Take one and pass the rest".

Replaced the breadplate by n^j /for a fixed j/ and the slices of bread by x_i^j /for i = 1,2,...,k/ we can similarly deal out the task of examining the bits, regarding whether $x_i^j = 0$ or $x_i^j = 1$, among copies of a cellular automaton module called, from now on, *MAXEL module*. See figure 3.2-1, and figure 3.2-1.

Data arriving - in a suitably represented form of signals - to E_2, flow along the DATA PATH /DTP/ to be controlled /kept or annihilated/ by the i-th information n_i^j about n^j. Data leave the unit at S_2 to enter the next MAXEL module U_{i+1}^j. The definition of the "information n_i^j about the data n^j" is, recursively,

$$n_{i+1}^j = n_i^j - 2^{k-(i+1)} x_{i+1}^j$$

with

$$n_o^j = n^j, \quad x_{k+1}^j = 0, \quad i = 0,1,2, \ldots, k.$$

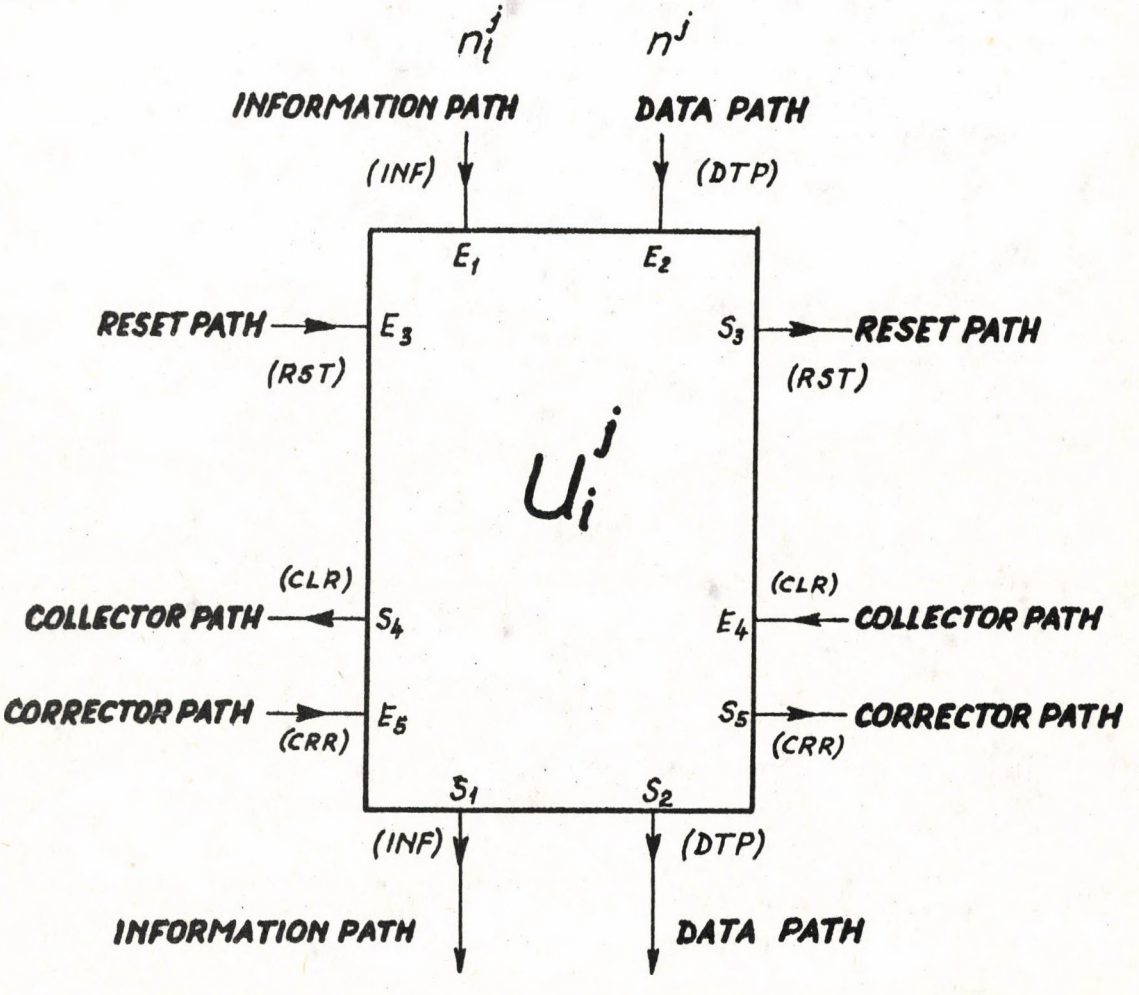


Figure 3.2-1

The i,j-th MAXEL module U_i^j

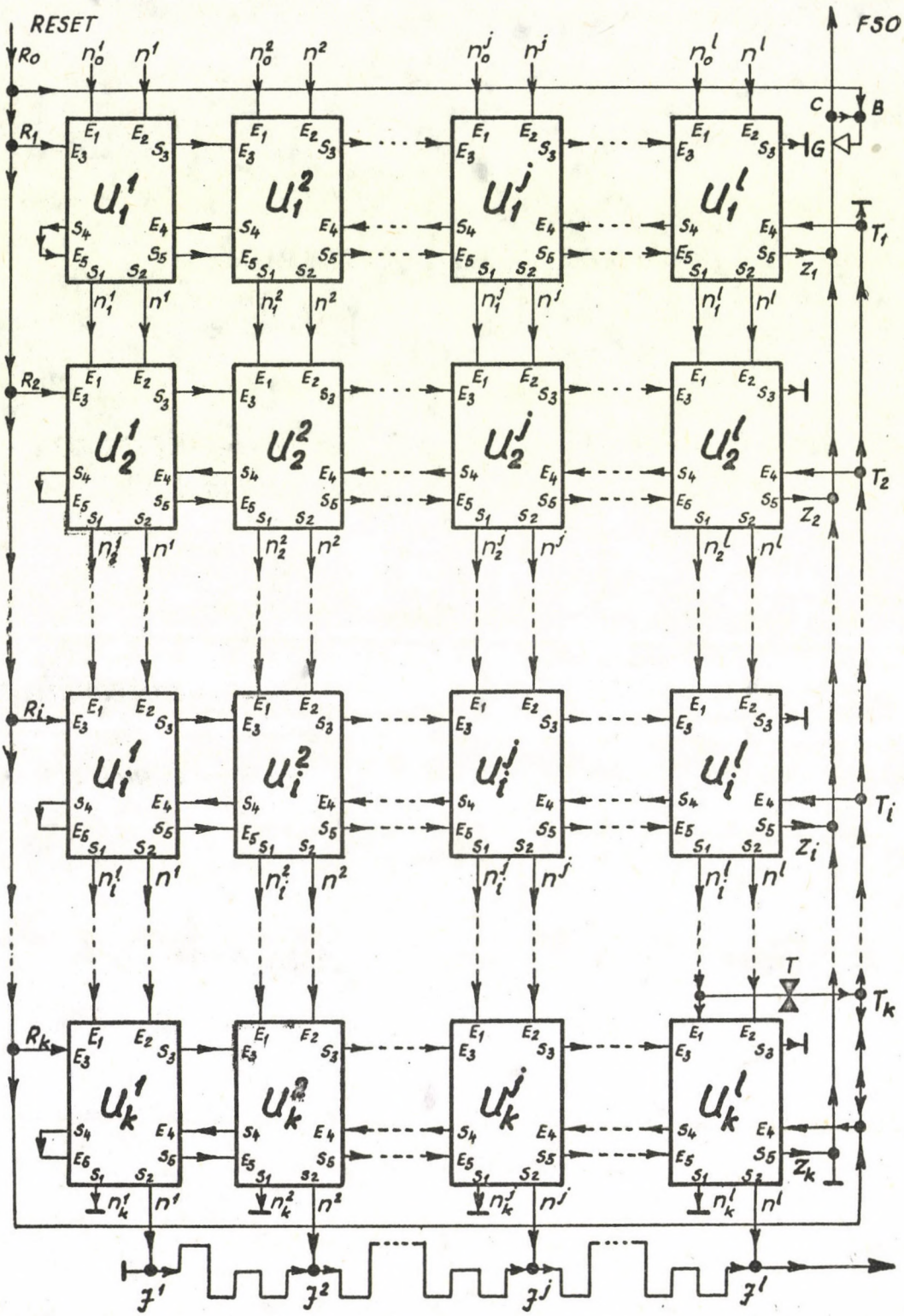


Figure 3.2-2

Modular implementation of the Bread-Dealing Algorithm

and, when implemented, the i leading zeros are suppressed /i.e. replaced by the cellular representative of the blank/.

Entering the module at E_1, n_i^j travels along the information path /INF/ to be "beheaded", somewhere in the inside of MAXEL and to leave it, in the form of

$$n_i^j = 2^{k-i}\, x_{i+1} + 2^{k-i-1}\, x_{i+2} + \ldots + 2^0 x_k,$$

with the i leading zeros suppressed. Thus, while n_i^j represented by a k-i signal string, n_{i+1}^j has only (k-i)-1 = k-(i+1) signals to represent the information about the original data. This is the information being the rest passed by the module while the "topmost slice of bread" i.e. the signal for bit x_i^j is taken out to be examined and used for control.

Next, the "allnought case" is to be somehow represented structurally. Unlike the cases that a bit x_i^j takes the value of 0 or 1 the allnought case depends not only on one bit, but rather on the whole system of bits.

So, unlike bitcases or bitevents /$x_i^j = 0$ or $x_i^j = 1$/ allnought cases or allnought events are not *local* events any longer, but rather, *global* events. It structurally implies that a new path is to be introduced connecting all the MAXEL modules U_i^j, with fixed i, to collect, bit by bit, the information about the bits x_{i+1}^j in a row. Let this path be called, for short, the Collector Path /CLR/. After having collected all the relevant information about the bitcases, travelling from right to left /see figure 3.2-2/ it makes a U-turn at the extreme left of MAXEL /after leaving U_i/ and, in the possession of the information about the allnought case, it is ready to provide correction for those modules which have been going to cancel an n^j because of its $x_i^j = 0$. From the entry point E_1 let the path be called Corrector Path /CRR/. The most convenient way for it to provide the correction signals for the modules is, that if allnought case has occurred there is a signal propagating along it and, if not, there is no signal at all.

Up to now, we dealt with all the four crucial paths DTP, INF, CLR and CRR. Theoretically, however, there remains an additional one to structurally represent the reset function. After having the maximum selection performed MAXEL's have to be *reset* to their initial states in which the modules are ready to accept the next task to accomplish. Thus a *Reset* signal /RST/ is necessary. Its only function is to reset the units one by one providing signals, conveniently, in a serial fashion. See figure 3.2-2. Starting from point R_o the reset signal duplicates at each point R_i, $i = 1,2,\ldots,k$ and enters the first module-column at points $/E_3/_i^1$, $i = 1,2,\ldots,k$, respectively. Then, with a delay, the reset signal descendants travel quasi-simultaneously along paths

$$/E_3/_i^1 \longrightarrow /S_3/_i^1; \qquad /E_3/_i^2 \longrightarrow /S_3/_i^2; \ \ldots; \ /E_3/_i^{\ell} \longrightarrow /S_3/_i^{\ell}$$

for $i = 1,2,\ldots,k$.

Finally, after leaving the last module, passing $/S_3/$, they are annihilated at the ends of the paths. At point M the reset signal enters to reset the collector's gating mechanism. Its detailed function will be clear later, automatically.

Now we can get an overview about the whole operation of the system. In the first stage one loads the information n_i^j, onto the information paths. As a result, a structural change takes place inside the device closing all the data paths except the one/s/ belonging to the maximal number/s/. This first stage consists of two substages. During the first substage the bit-dealing process is carried out i.e. all the kx bits x_i^j reach their modules U_i^j to be examined. After having the modules in the last row, i.e. modules U_k^i with $j = 1,2,\ldots\ell$, been passed, information signals n_i^j are annihilated at the ends of the information paths, /in fact prior to this, inside the U_k^i by the last "beheading"/ the last path, containing n_{k-1}^{ℓ}, excepted. It consists of only one single signal and triggers, through $N \longrightarrow T \longrightarrow T_k \ /E_4/_k^{\ell}$, the collector signals in the order of T_k, T_{k-1}, T_{k-2}, ..., T_1. T is a signal transformer to

produce an adequate form for the collector signal. Thereby the second substage of stage one is started. After having zig-zagged through the modules the corrector signals will be collected again through Z_k, Z_{k-1}, Z_{k-2},...,Z_1 to produce a signal FSO at point A indicating that the First Stage is Over. Gate G ensures that only the first signal reaches point A the rest being annihilated by G. The varying number of the surviving corrector signals, owing to the variable allnought cases, could cause trouble that's why it is uniformed by G.After a certain delay signal FSO can be made to reach a peripherial device where data n^j are stored. FSO can trigger it to be loaded in through the information paths to commence the second stage of operation.

Having finished with the second stage, the surviving n^j-/s/ /eventually identical copies of the maximal n^j/ will reach the Join Path $J^1 \longrightarrow J^2 \longrightarrow ,..., \longrightarrow J^{\ell}$ at the MAXEL's bottom. Owing to the delays inserted between J^j and $J^{j+1}$the surviving bit-strings won't collide and, one by one, will proceed to M to leave MAXEL.

After this the user only has to do is to reset MAXEL. This, if necessary, can be automated but it is more economical to refer to the user as regards providing a single reset signal at R_o. Incidentally, the reset signal opens the locking mechanism at G through B.

CHAPTER 4.

CELLULAR BAKGROUNDS

4.1 Historical overview

Cellular automata studies began with von Neumann's classic lecture at the Hixon Symposium in 1948. /von Neumann, 1966/ Sixteen years later, J. von Neumann's posthumus book, edited by A.W.Burks, with a fairly detailed system of ideas concerning cellular automata had been published. Von Neumann

himself has designed a self-reproducing cellular automaton and many useful special automata /organs such as pulsers, decoders, crossovers etc./.

Von Nemumann's cellular space was characterized by the following features. /This consize description of von Neumann's space is due to E.F.Codd, 1968; p2./ Our italics show the essential points where new developments started.

1/ An *infinite plane* is divided up into *squares*.

2/ Each *square* contains a copy of *the same* finite automaton. The square together with this automaton is called a cell.

3/ Associated with each cell is its neighbourhood consisting of itself together with its *four immediate*, nondiagonal neighbours.

4/ The state of a cell at time t+1 is uniquely determined by its *neighbourhood state at time* t, together with the *transition function* f of the finite automaton.

5/ The finite automaton associated with each cell possesses *a distinquished state* V_o called the quiescent state, such that

$$f/V_o,V_o, \ldots, V_o/ = V_o.$$

6/ At each time step *all but a* finite number of cells are in the quiescent state.

7/ The number of distinct states for the finite automaton associated with each cell is *29*.

8/ A particular transition function f *is specified* and shown to yield certain *computational and construction properties*.

Let our overview be centered around these eight points by indicating the major changes and developments achieved in the recent ten-some years.

ad 1.:

Along effecitivity oriented studies: Dettai /1974/, Doman /1974/, Doman /1975/, Fay /1974/, Fazekas /1975/, Golze /1972, Szőke /1975/, Takács /1973/, Toffoli /1975/ infinity postulate is dropped. The edge of the cellular space plays a very important interfacing role. Fazekas /1975/ has designed a cellular automaton called RETINA, in a cellular space CODD-ICRA, which can read the information from the edge of the space and transfer it to the inside to construct automata. Takács /1973/ has designed a bootstrap

for cellular automata which is to be attached to the edge of a /CODD-ICRA/ cellular space in order to develope the most primitive components of cellular automata. Even RETINE's parts are to be first bootstrapped into the space. Of course, simulation studies are most strongly concerned with finite spaces: See, also for a survey of cell space simulations, Legendi /1975/. As for the plane, it is superseded at Doman /1974/, and /1975/ by a three dimensional cellular space where squares are replaced by cubes.

ad 2.:

Strictly speaking the edge-cells are *not the same* as the inside cells.

ad 3.:

The number of neighbours /as well as of the states/ are widely varying. A summary can be found in SMITH, III /1969/. Conway /1970/ invented the "game life", being a wide class of cellular studies, where 8 neighbours are favourized.

ad 4.:

Needless to say, cellular space is a clocked system. If we drop the property of being clocked we get the idea of cellular array a branch again grown out from cellular studies. A fresh and nice treatise on cellular arrays can be found in Ippolito /1972/.

In cellular arrays where cells are possessing only combinatorial rather than sequential logic, it is not true any longer that the state of a cell at time t is uniquely determined by its and its negihbours' last states.

As for the transition function f, there are several ways of defining it. One extreme case is when it is defined completely logically i.e. by rules. Von Neumann has defined his transition funciton this way. /Von Neumann, 1966/. The other extremity is if the transition function is defined completely by its truth-table. Codd /1968/ has defined his transition function nearly that way but his table, for the partial transition function, has been completed by some rules. Codd's truth-table /found by an interactive computer trial-and-error technique/ was then superseded, step by new rules. /Fazekas, 1975, Szőke 1975/. By this "regularization" new features have been introduced regarding Codd's automata.

ad 5-6.:

A "second quiescent state" was invented by Fazekas /1975/ when elaborating this "blueprint shift technique". One brings the cellular space /CODD-ICRA/ into the all-one state i.e. each cell is in state 1 /the second quiescent state/ then the blueprint of an automaton /in staked form,

i.e. its 0-1 configuration/, put row by row to the edge of the space, can be rolled in, or shifted in on this all-one carpet.

ad 7.:

The number of the cellstates of a von Neumann's cell has been supplemented by a new "crossover state" by Dettai /1975/. Codd /1968/ used eight states, Lindenmayer has initiated a cellular space with two cellstates, see Herman-Rosenberg /1975/. The four-/+1/ neighbour two-state /0,1/ space with the transition function,

$$f(x_1,x_2,x_3,x_4,x_5) = x_1+x_2+x_3+x_4+x_5$$

$$(0+0 = 1+1 = 0,\ 0+1 = 1+0 = 1)$$

is called Lindenmayer space by us, the ICRA TEAM and intensively studied by Martoni /1975/. Doman /1975/ works with a cell having more than five thousand states. Codd's cells, as well as Lindenmayer's cells are automata without output, while Doman's, like von Neumann's, are automata with output i.e. it sends different signals to different cells /neihgbours in different directions/.

ad 8.:

Up to Codd /1968/ all the designs have only been aimed at theoretical elucidations of certain cellular automata properties rather than at more practical engineer minded constructions.

Codd was the first who put in action the way of viewing cellular automata from users' aspects. He invented /by an interactive computer technique/ a considerable number of elementary cellular automata, called components, such as the *sheathed paths*. A sheated path is a row of cells in state one with layers on their right and left sides like this:

```
222222222222
111111111111
222222222222 .
```

Along the paths, signals can be sent in the form of a pair of adjacent cells in states 0 and S, respectively, for S =4,5,6 and 7. By adequate manipulation of the signals [04], [05], [06] and [07] one can

- *move* the head of the paths in four directions /right, left, backward, forward/
- *read, write and earse* cellstates 0 or 1.

This way several more complex structures can be built up /see the next paragraph/ but the destruction of these automata was yet to be solved. Golze /1972/ dealt with destruction and Szőke /1975/ made a step forward as far as the effectivity of and the compatibility with CODD-ICRA

is concerned.

In the next paragraph a brief account is given for Codd's space. As for the cellular researches carried out till 1968 A.W.Burks /1968/ is referred to. For the later story ICRA TEAM is going to give a fairly detailed account. At present Aladyev's /1974/ survey can be considered as the best.

4.2 Codd's space

By Codd /1963/ a cellular space has been elaborated with the aim of understanding the transition function. The principal and crystal clear presentation, the introduction of new basic cellular automata concepts, the clarification of new design philosophies and techniques have been, naturally, preferred to design economics and effectivity. This cellular space has the features below. /Our account is centered around the points where some expansion was made to yield CODD-ICRA space/.

1/ An infinite plane is divided into squares.

2/ Each square contains a copy of the same outputless automaton clocked simultaneously.

3/ Associated with each cell is its neighbourhood consisting of itself together with its four immediate, nondiagonal neighbours.

4/ The state of a cell at time t+1 is uniquely determined by its and its neighbours' last states specified by the transition function f.

5/ The cell /short for "the finite automaton associated with each square"/ possesses a class P of states p_i called passive states such that

$$f(p_1,p_2,p_3,p_4,p_5) = p_i$$

for $p_i \in \{0,1\} \in P = \{p_1,p_2,p_3,p_4,p_5\}$,

$$i = 1, 2, 3, 4, 5.$$

6/ At each time step all but a finite number of cells are in state 0, the quiescent state, for which

$$f(0, 0, 0, 0, 0) = 0.$$

7/ The number of the cellstates is 8.

8/ A particular partial transition function f is specified by two truth tables and two rules. By this, several phenomena can be brought about and a number of cellular automata components can be produced.

9/ An interactive simulation technique has been developed for establishing the desired transition function by strengthening and augmenting designer's heuristic.

10/ A self-reproducing computational universal cellular automaton has been constructed.

From our design's point of view mainly paragraph 8 is the more interesting. Codd's transition function is defined by his "long table" pp. 67-68 in his book. There were two rules attached to it.

The first rule is the "rule of small terms" i.e.:
The center cell remains unchanged in any neighbourhood where the cells are in "small states" i.e. state 0, 1, 2 or 3. There are a few /nine/ exceptions to this rule. They are contained in the "short table" /p. 66./.

The second rule is the rule of rotationsymmetry. It means that the next state of a cell is the same if its neighbourhood is rotated around it clockwise by a right angle. Thus the long table worths nearly four truth tables. Not exactly, since there are neighbourhoods whose three rotations /by 90 degress/ yield less than three new neighbourhoods. For instance the rotation of a neighbourhood consisting of cells all in the same state is immaterial. Thus out of the possible $8^5 = 32768$ term there remains not $8^5/4 = 8192$ but 8352 rotationsymmetric /different/ cases.

By this rule it is enough to put only the "cyclominimal" term into the truth table, i.e. those having the minimal numeric value among the four ones received by clockwise right angle rotations.

Codd's transition function is a partial function i.e. it is not defined everywhere. Out of the 8352 possible /rotation symmetric/ cases only 512 terms are tabularly defined. The rule for small terms takes care for 280-9 /rotation symmetric/ terms, therefore, altogether in

$$512 + (280-9) = 783$$

cases is the transition function defined. It seems, that this function is quite unspoiled for it is exploited only up to a degree of

$$\frac{783}{8352} \approx 9\ \%$$

The most important /regarding our design at least/ cellular automata component invented by Codd are

- The sheated path along which signals can propagate.
- Branching and looping paths where signals can be reproduced, transformed and annihilated.
- Gates by which signal propagation can be controlled.
- Path-ends by which one can read, write and erase cellstates 0 and 1.

Codd has introduced three phases of construction: *staking* /i.e. establishing the 0-1 configuration for the paths/, *sheating* i.e. providing two layers of cells all in state 2 along both sides of the path's core, and *activating* i.e. bringing the gates /cells in the sheaths of paths/ into active state /state 3/ where necessary.

The elementary components above can be combined to construct components of higher level of organization.

These are eg. the one-way locks, 07-transformers, signal sources, echo discriminators, crossovers, decoders, etc.

In the recent years, now that hardware implementation is very close to reality, the interest in cellular automata studies is rapidly growing. Just to mention some outstanding events we refer to three recent conferences. See Riguet /1974/ Herman /1974/ and Lindenmayer /1975/.

4.3 CODD-ICRA space

Three years later after Codd's book a team was formed in Hungary /see ICRA TEAM, 1976/ venturing upon the implementation and effectivization of Codd's space. This team had realized that in the near future a cell in a form of an LSI chip would become a reality. Nowadays, at the end of 1975,

no doubt, it is indeed the case. An INTEL 8080 microprocessor /"computer on a chip"/ is fare more complex than a CODD-ICRA cell would be.

Of course, the implementation of the cellular space in a convenient form of hardware, is by no means the crucial question anyway. We think that, rather, design effectivity is the keyword. The counterpair of *programming* in cellular space, is: *designing* some machine. Or, with Codd's words: "reorganizing the computer in a problem oriented way".

Now, as for programming, we have lots and lots of fine techniques, ample experiences, sophisticated procedures, textbooks, manuals, institutions and a hundred and one other things to serve traditional computer science. Even the most exact mathematical tool, axiomatization, is entering the computer science. /For this, see Hoare's /1969/ very interesting paper/.

With these underlying ideas we have embarked upon the work of putting cellular automata into practical use. Codd's space has been chosen for many reasons. The most important ones are:

- its didactic clarity in its presentations;
- its unique technique of interactive computer usage to invent and test new cellular automata;
- its being very engineer-minded.

The work has begun with the crossover problems. To cross two paths took above 3000 cells. Having introduced tensome new terms, without having conflicted with the previous constructions of Codd, Dettai /1974/ invented a *crossover* /for signals 06 and 07/ with as few cells as 25. At the same time *locks* /taking hundreds of cells previously/ have become constructible with two cells; *signal source* with four cells /rather than thousands of cells/ and other minor /but useful/ things have been constructed.

To avoid misunderstandings we stress here that Codd himself has not dealt with effectivization at all, rather, attemped to lay down the new principles in a possible simple form. Had be tried to design cellular automata more effectively

the principal values of his work would have had certainly suffered.

Then Fazekas came /1975/ and exploiting Dettai's results to the last he constructed a great deal of new components and discovered a few new phenomena.

Just to list, by names, his main parts:

- *growing trees*: treelike paths without any previous sheathing, to solve the problem of parallel readwrite erase;
- *monostable gates*;
- *reading from head* /an opposite path's end/;
- *discriminators*;
- *selectors*;
- *storing by "bubbles"*: a bubble being a cell in state 0 in the core of a path;
- *phase converter* to produce signals following each other both with even and odd lag;
- *new* phenomena of *signal collisions*, etc.

Making use of Dettai's parts, as well as his own, he was able to construct a device, called RETINA, by which one can transfer any information from the edge of the cell space to the inside. Also, a "blueprint shifting technique" was invented by which, in a true parallel fashion one can roll stakings /on the "carpet of ones"/ into the inside of the space.

A two semester lecture was given at Budapest Eötvös Loránd University by Fáy (the author) /1975/ where a fraction of these results had been systematized. Takács has designed a bootstrap for the cell space [Takács, 1973].

Szőke /1975/ discovered some rules in Codd's transition function eg.

$7 \longrightarrow 0$ if there is an odd state in the neighbourhood,

$7 \longrightarrow 1$ if there is no odd state in the neighbourhood.

Also, she elaborated destruction in the modified Codd' space called CODD-ICRA /Iterative Cellular Realization of Automata/

Fay /1974/ implemented an edge-notched card selecting system in CODD-ICRA realizing that a data base management technique DSL ALPHA, /invented by Codd incidentally/, can be simulated this way Takács /1974/ has written a program in DSL ALPHA to get a step further in conceptual data processing by cellular automata. Legendi /1975/ has developed a simulation language called CELLAS by which all the newly introduced components can be tested. Huszár /1975/ has designed a quite universal equipment for testing implemented cells electronically. Bagyinszki /1975/ suggested an original way of parallel computation in cellular media through a residue number system. CODD-ICRA has enjoyed twofold challenges. Dettai /1975/ worked out the implementation of the von Neumann cell, added the 30-th state and after this it turned out that the cell is far more simple to implement than Codd's eight-state cell even after its transition function has radically been regularized by Szőke /1975/. On the other hand, Doman /1974/ invented a three-dimensional cellular space with cells more than five thousand states. See also Doman /1975/. A Doman's cell seems to be easier to implement electronically even than the von Neumann's cell. True, self-reproduction is not aimed at, rather, practical effectivity is preferred to it.

As for the more formal definition of CODD-ICRA its transition function's truth table can be seen in Table 4.3-1. In addition to this, there are the following rules and postulates.

1/ CODD-ICRA transition function $F(X_1,X_2,\ldots,X_5)$ is a mapping from $S_1xS_2xS_3xS_4xS_5$ to S_1 where $S_1 = 0,1,\ldots,7$ the set of the center cell's state $S_i = 0,1,\ldots,7$ the set of the center cell's neighbours' states. $i = 2,3,4$ and 5 refers to the right /eastern/, bottom /southern/, left /western/ and upper /northern/ neighbour, respectively.

X_i S_i $i = 1, 2, \ldots, 5.$

2/ F is rotationsymmetric, i.e.:

$$F(X_1,X_2,X_3,X_4,X_5) = F(X_1,X_5,X_2,X_3,X_4)$$
$$= F(X_1,X_4,X_5,X_2,X_3)$$
$$= F(X_1,X_3,X_4,X_5,X_2)$$

3/ F satisfies the "rule of high terms" i.e.:

6,7	0	iff there is an odd state in the neighbourhood
6,7	1	iff there is no odd state in the neighbourhood
4,5	0	iff there is a cell in state 1 in the neighbourhood
4,5	1	iff there is no cell in state 1 in the neighbourhood

4/ F satisfies the "rule of passive neighbourhood" i.e.:

$$F(X_1,X_2,X_3,X_4,X_5) = X_1$$

Unless otherwise stated by the rule of high terms or by the following truth table: /it differs from the usual CODD-ICRA table by the term 016761 needed for reverse lockpair activation, cf. Ch 9./

Table 4.3-1. *Truth table for CODD-ICRA transition function*

000062	012141	016221	102535	112255	122366	201071
000073	012151	016261	102616	112266	122377	201171
000153	012161	016621	102626	112277	122434	201423
000162	012171	016661	102636	112424	122444	201711
000252	012227	016761	102727	112434	122535	202060
000262	012231	017171	102737	112445	122555	202073
000422	012241	017221	103424	112525	122636	202513
000513	012251	017271	103434	112535	122666	202063
000612	012261	017721	103525	112556	122737	203073
000622	012271	017771	103535	112626	122773	207111
000662	012326	022262	103626	112636	123244	211171
001062	012351	022662	103636	112666	123255	212323
001073	012421	100040	103727	112727	123266	223243
001162	012441	100066	103737	112737	123277	223253
001262	012521	100073	104110	112774	123344	223263
001612	012531	100140	106066	113437	123355	223273
001622	012551	100166	106116	113537	123366	300022
001662	012621	100244	106166	113637	123377	300061
002062	012661	100266	106216	113737	123434	300251
002073	012721	100366	106226	114224	123535	300260
002131	012731	100410	106266	114245	123636	300270
002262	012771	100525	106616	114425	123737	300421
002612	013131	100616	111140	115225	124244	300620
002622	013221	100626	111156	115256	124334	300720
002721	013241	100636	111166	115526	125255	301022
003631	013421	100666	111167	116166	125335	301030
006112	013521	101040	111244	116226	126266	301064
006212	013621	101055	111255	116266	126336	301077
006222	013631	101066	111266	116626	127277	301111
006262	013721	101072	111277	116663	127337	301620
006612	013731	101140	111424	117177	133344	301720
011162	014221	101166	111525	117277	133355	302610
011241	014241	101266	111626	117274	133366	302710
011251	014321	101410	111663	117724	133377	311111
011261	014421	101616	111727	117773	200060	312322
011271	015221	101626	111773	122244	200071	323242
011421	015231	101666	112144	122255	200171	323252
011521	015251	102266	112155	122266	200253	323262
011621	015351	102424	112166	122277	200423	323272
011662	015521	102434	112177	122344	200711	
012121	016161	102525	112244	122355	201060	

example: 117724 means F(1,1,7,7,2) = 4

Out the components characteristic to ICRA only two are extensively used along our design. These are the crossover /for signals [06] and [07] only/ and the locks. Figures 4.3-1,2,3 and 4-3.4 explain their working.

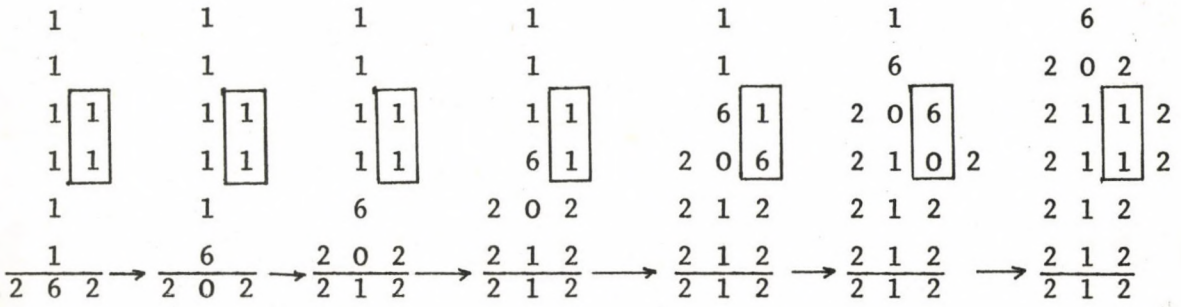


Figure 4.3-1.

The staking and sheathing of a lock

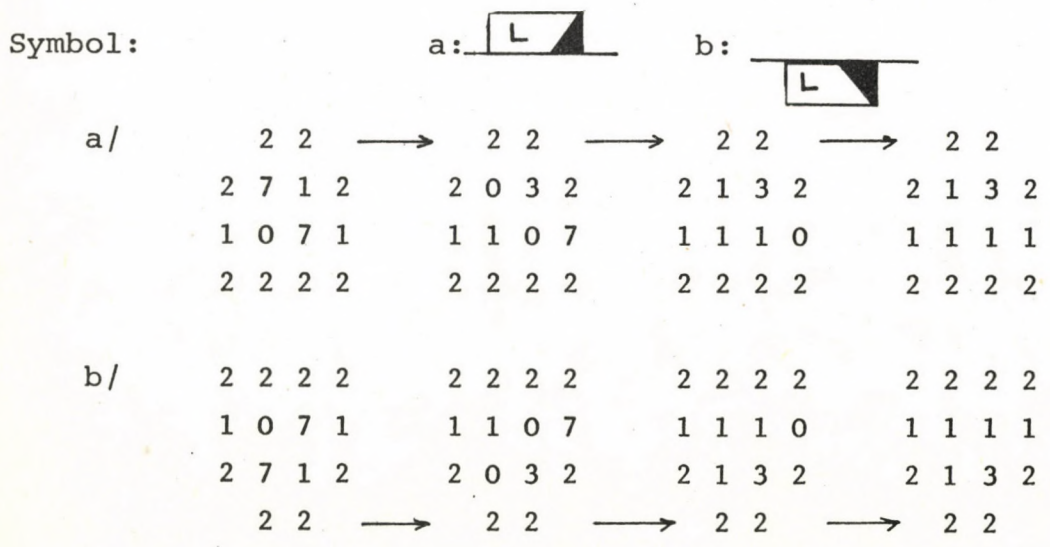


Figure 4.3-2.

The activation of a lock /two possible version a. and b./

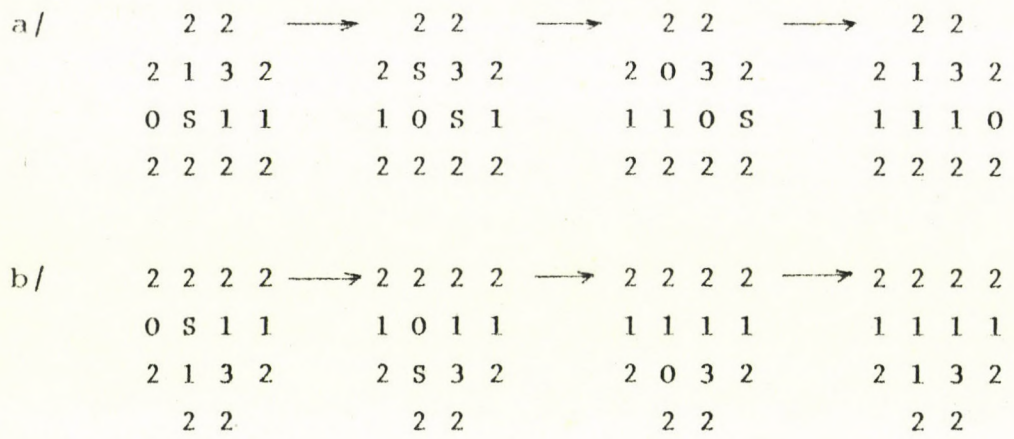
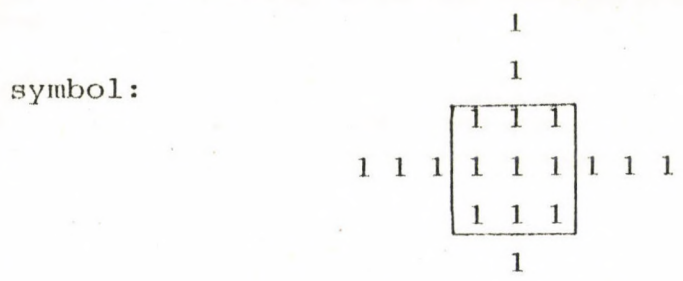


Figure 4.3-3.

The operation of locks for S = 6 or 7



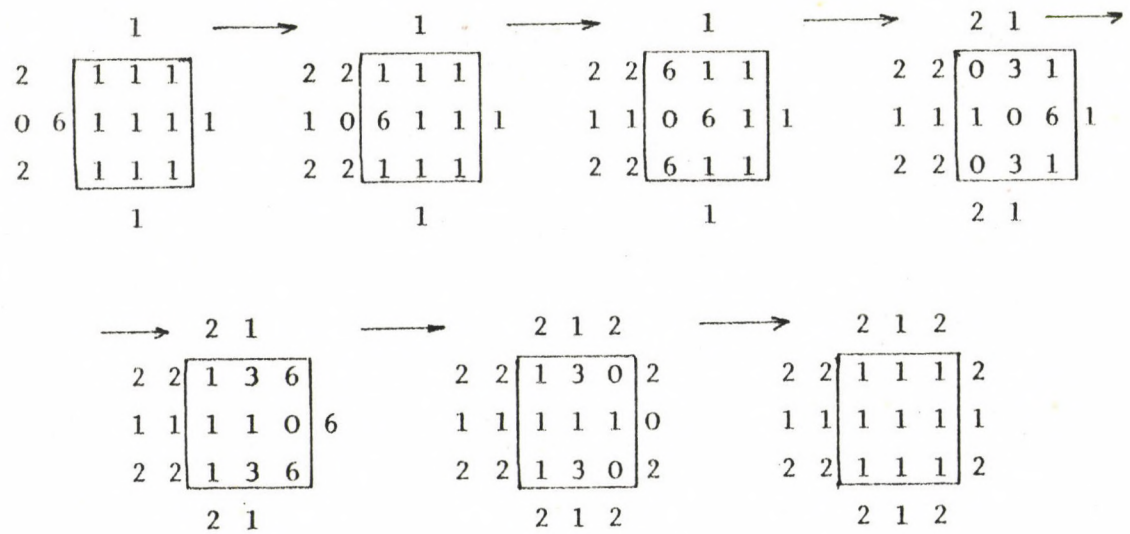


Figure 4.3-4.

The staking and sheathing of a crossover

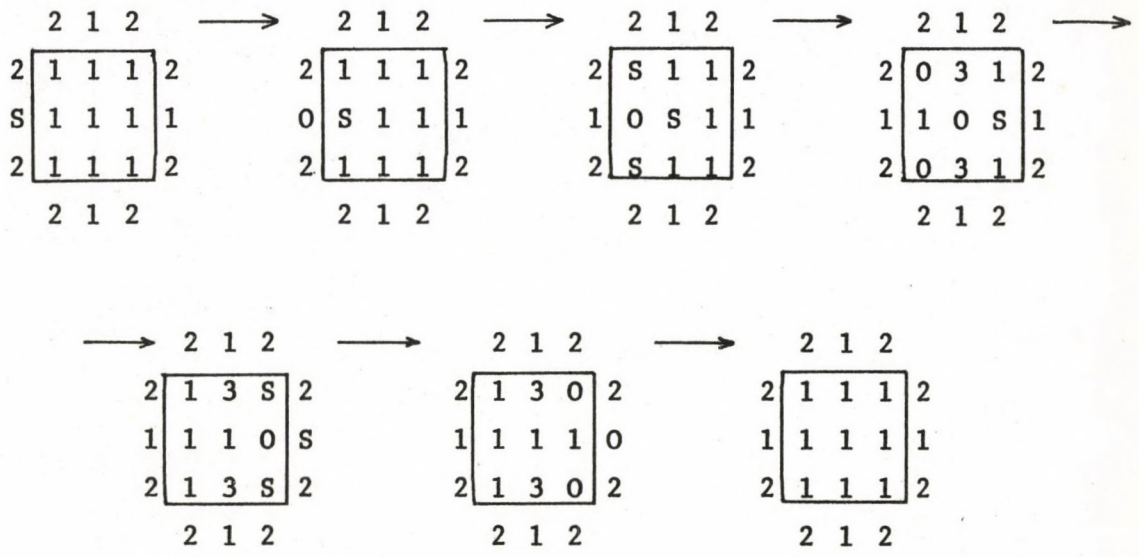


Figure 4.3-5.

The operation of Dettai's crossover S = 6,7.

CHAPTER 5.

CELLULAR ASPECTS OF THE PROBLEM

5.1 An intuitive way to approach the MAXEL module

In Ch.3. a phenomenological /or functional system design has been carried out. We didn't care about the structural elements implementing the functions to be performed. In fact we have considered as few characteristic features of CODD-ICRA space as possible. Essentially we made use of the possibilities to construct,

- *paths*, along which *signals*, representing data can propagate;
- *gates* by which signals can be *annihilated*;
- *pathforks* where signals *duplicate*

- *signal transformers* to convert signals into each other;
- *locks* for implementing *directed paths*;
- *crossovers* providing means for crossing paths.

Now that we are aware of the exact mechanism of the elementary structures above we can make an attempt to combine them in order to implement the functions talked over in Ch.3.

Let us start with a black box and try to unwrap it successively. In figure 3.2.-2 one can see the MAXEL module's outside features with the stress placed on its paths. Let us try first to peel out the gates controlling the paths.

For the sake of brevity let the signal, representing the first bit of the information input n_i^j of U_i^j /see figure 3.1-1/, be denoted by x and any of the rest by y.It is clear that the data flowing along the data path can be controlled by a gate G_i^j, placed on the righthand side of the DTP directed downwards. /See figure 5.1-1/. This gate G_i^j has to be closed if $x = x_i^j = 0$ has been the case except when all the other bits with the same local value /i.e.: i/ are also zero, i.e. $x_i^1 = x_i^2 = \ldots x_i$. In this "allnought case" a correction is to be made which is very easily implemented by a branching off from the CRR /corrector path/. This is managed by forkpoints F_1 and F_2. On the other hand, in order to provide information about the allnought case one has to place a gate /G_2/ on the right hand side of the collector path /CLR/ directed from right to left. This gate G_2 must be open /OFF/ whenever $x = x_i^j = 0$. If /and only if/ all the gates G_2 of the U_i^j-s in a row /i.e. U_i^1, U_i^2, ... U_i/ are off can "allnought" be the case. This way there will be no obstacle in front of the collector signal entering the unit at E_4 to travel on and, making a U-turn after leaving the leftmost unit U_i^1 of MAXEL, produce the correction signal for turning all the gates G_2 off.

Incidentally, we can see that the following data - signal correspondence will suit the task:

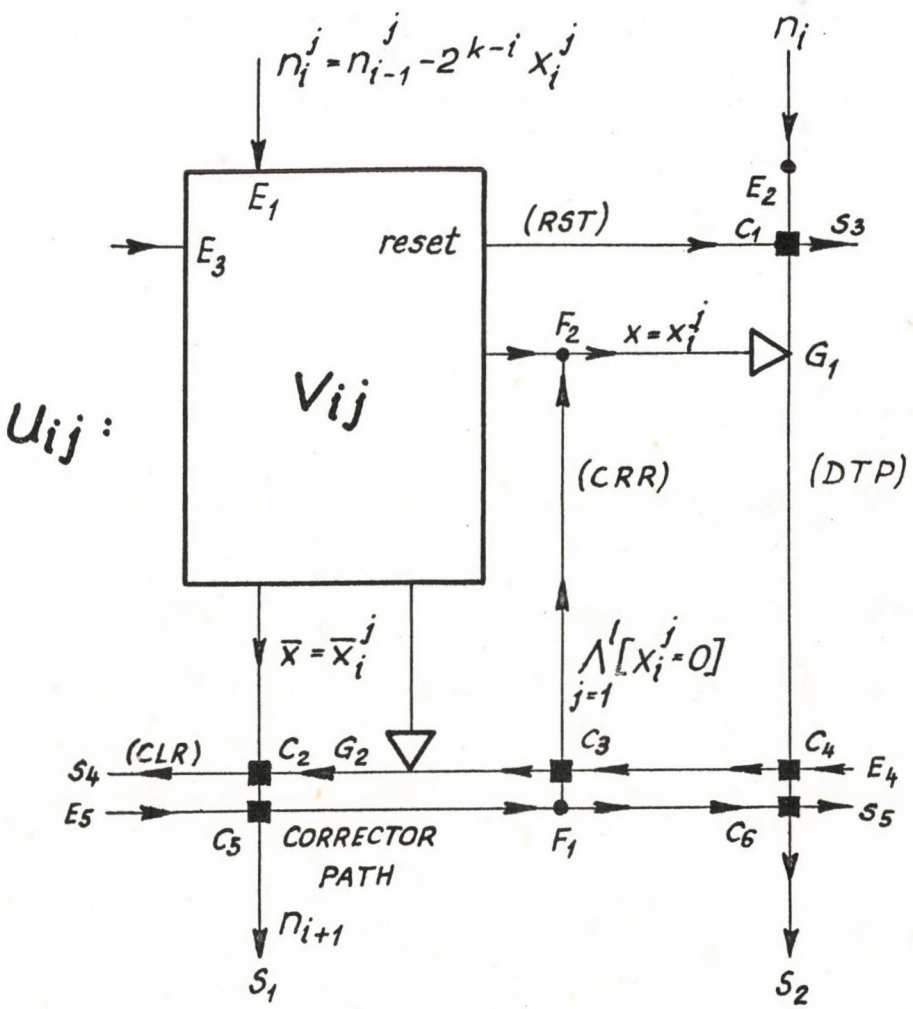


Figure 5.1.1

Unwrapping the MAXEL module U_i^i *first step*

$$x = [07] \quad \text{if} \quad x_j^i = 0$$

$$x = [06] \quad \text{if} \quad x_i^j = 1$$

collector signal $= [07]$.

Moreover, G_1 is on iff $x_i^j = 0$ except $\bigwedge_{j=1}^{\ell} [x_i^j = 0]$

G_2 is off iff $x_i^j = 0$

One can also see that altogether 10 elementary structures /cellular automaton components/ have been to be introduced. These are:

6	Crossovers:	$C_1, \ldots, C_6$
2	Forkpoints:	F_1, F_2
2	Gates:	G_1, G_2
10	elementary structures altogether.	

As a second step, in unwrapping the black box of U_i^i, let us concentrate on the yet unwrapped core V_i^j of U_i^j. See figure 5.1-2.

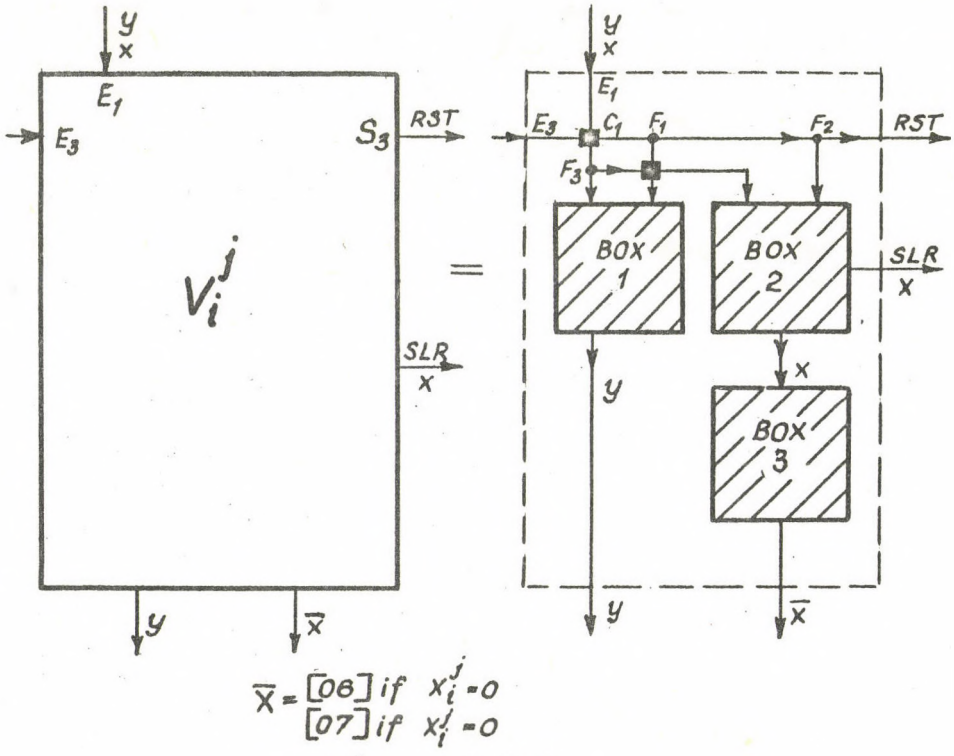


Figure 5.1-2.

Unwrapping the MAXEL module, second step

It seems to be useful to specialize the tasks. Box 1 kills the first bit /'s representative x of/ x_i^j, Box 2 kills the rest while Box 3 inverts x. Five new parts have been introduced at this step:

2	Crossovers:	C_1, C_2
3	Forkpoints:	F_1, F_2, F_3
5	new parts altogether.	

As for the boxes they are easily unwrapped one by one. See figures 5.1-3,4,5.

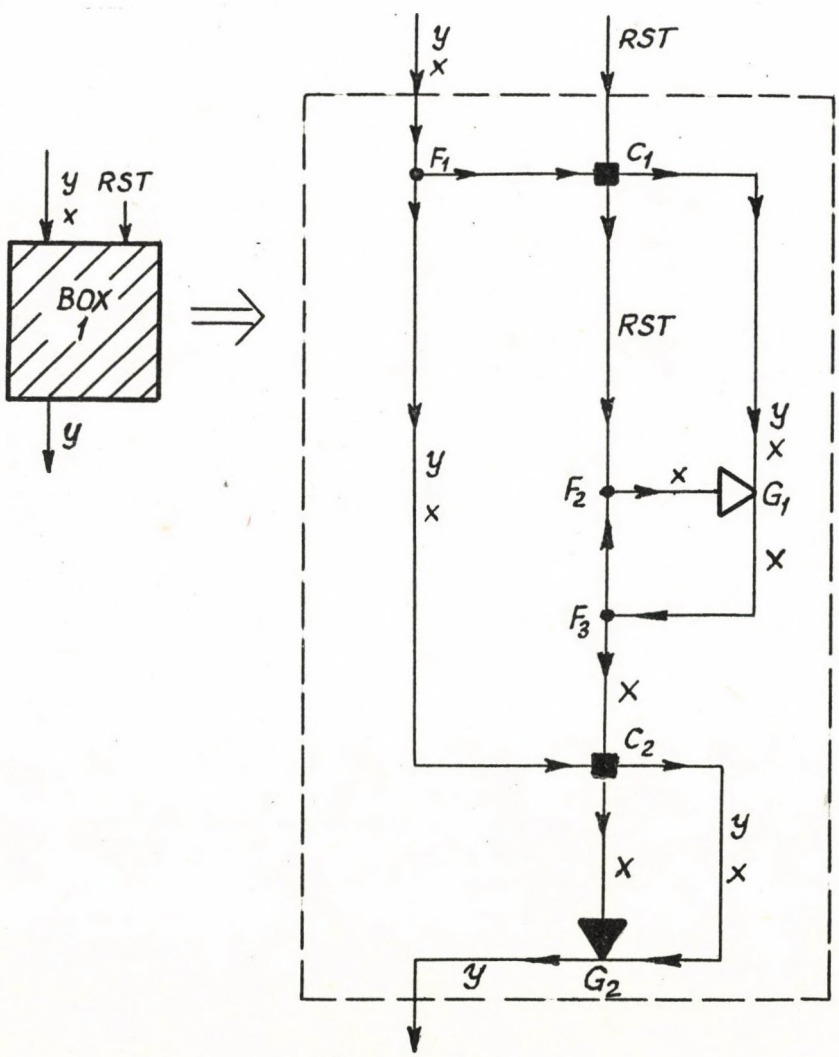


Figure 5.1-3.

Unwrapping the MAXEL module, third step

In essence, the network in figure 5.1-3, performs the function set before. Gate G_2, being on at the beginning, is turned off then after signal x arrives to G_2 along path $F_1 \longrightarrow C_2 \longrightarrow G_2$ is turned on then, provided part $F_1 \longrightarrow C_2 \longrightarrow G_2$ is shorter than path F_1-C_1-G_1-F_3-C_2-G_2, permitting the rest y to go but killing signal x. So, gate G_2 must be normally on.

Meanwhile G_1 acts as a resettable lock inhibiting the second signal /y/ to pass. Of course, the first signal in the control path of G_1 must be [07] to turn it on.

There remain four things to be cared for,

First: G_2 is to be normally on;

Second: to provide for G_1 only signal [07];

Third: to prohibit that signals trespass the reset path;

Four: to prohibit that reset signals trespass anywhere.

By placing some locks, all these minor gaps are filled. See figure 5.1-4. Inevitably, these newly placed locks $L_1,\ldots,L_6$ give rise to a new trouble, namely, to activating problems. During activation L_4 won't stop the unforeseeable perhaps unrepairable troubles, /signal collisions, crossover blockings etc./. To prevent this, a newer gate is to be placed provisorily between L_4 and C_1 branching off at F_4.

Seemingly, lockpair L_5, L_6 is redundant, for signal x is transformed into [07] at L_1,L_2 anyway so G_2 just like G_1 receives a signal [07] without making use of lockpair L_9, L_{10}. However, when resetting the module by a reset signal r = [06], while G_1 is turned off, G_2 won't be turned on unless r = [06] is transformed into r^t = [07].

That's why L_5, L_6 is put there.

By this the unwrapping process for BOX 1 is finished since no black spot remained to clear up. Our result is that BOX 1 contains altogether 13 cellular parts such as

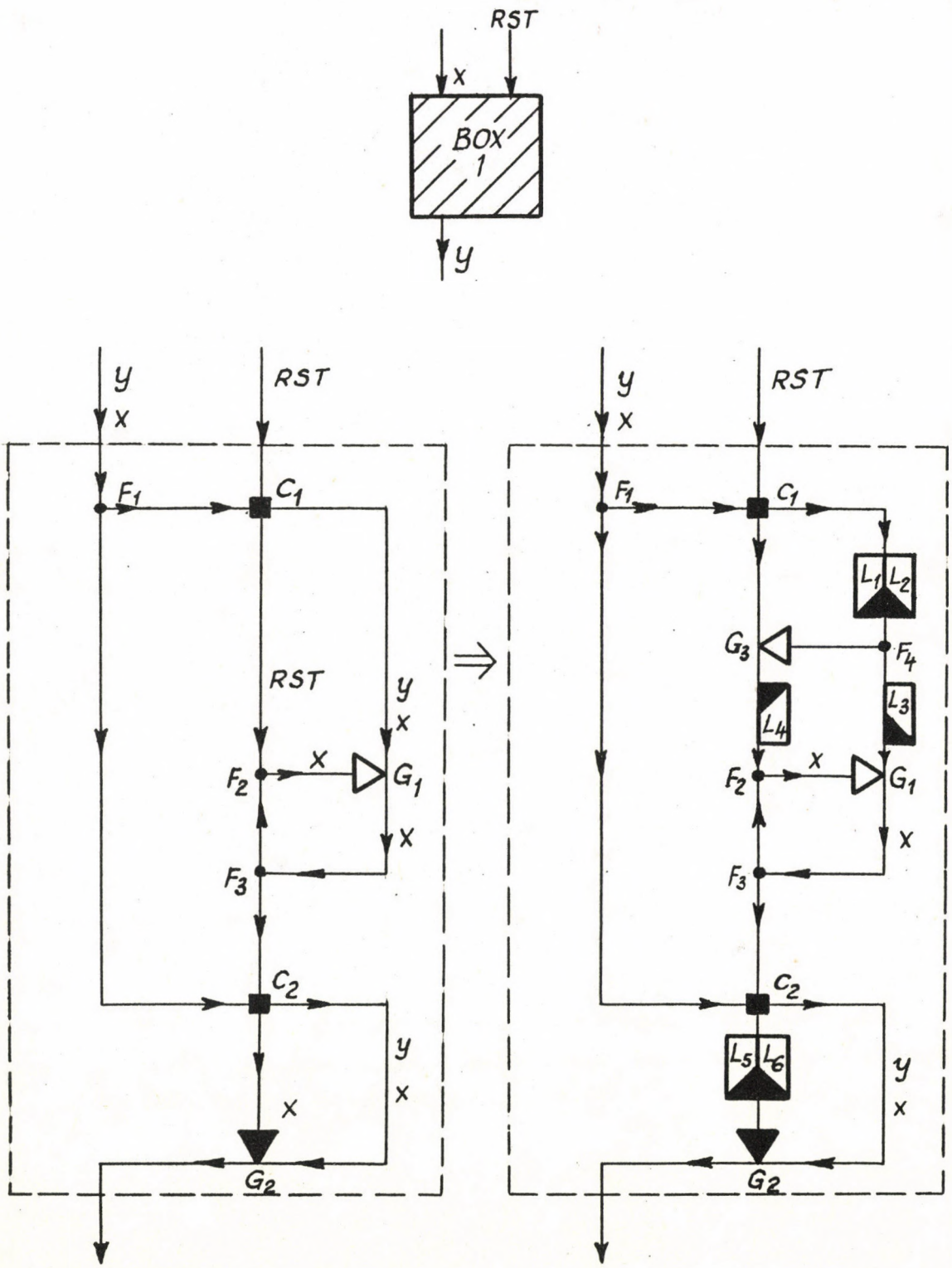


Figure 5.1-4

Unwrapping BOX 1 of the MAXEL module
/cf. fig. 5.1-2/

2	Crossovers:	C_1, C_2
4	Forkpoints:	F_1, F_2, F_3, F_4
3	Gates :	G_1, G_2, G_3
2	Locks :	L_3, L_4
2	Lockpairs :	L_1L_2, L_5L_6
13	parts altogether.	

It is quite typical and remarkable, in cellular design, that very useful byproducts are yielded to be utilized somewhere else in the main design. Here, first of all, we have gained a "single x" or "locked x" by which it is very easy to control again the information path to produce the "only x" signal for the SLR path in figure 5.1-2. In other words a resettable locking mechanism /like G_1-F_3-F_2-G_1, in figures 5.1-3,4/ in BOX 2.

Taking this advantage into consideration let us drain path C_1-F_4-G_1-F_3 at a point between G_1 and F_3 on figure 5.1-4. Introducing this path into BOX 2 one can unwrap BOX 2 in one step. See figure 5.1-5.

Gate G_1, fed from BOX 1, is normally off. After allowing x to pass, it is turned on by the locked and delayed signal x. Thus y /i.e. everything following x/ is annihilated. At point F_1 reset signal enters to reset BOX 3 and the gate placed on the end of path SLR /*Selector* path/. At F_2 SLR branches off. Of course, again, some minor design questions have to be cared for. Thus it is to be prohibited that the reset signal trespass the information path and, conversely that information enters the reset path. To this end lock L_1 and lock L_2 are to be placed. Also, as in the case of BOX 1, to annihilate the activation signal, gate G_2 is to be provided controlled conveniently from the data path. Similarily, lock L_3 prevents the corrector signal to enter BOX 2.

Now there remains only to unwrap BOX 3, the inverting unit. /See figures 5.1-2 and 5.1-6/. As it is wellknown, inversion is most conveniently performed by the aid of a clock signal.

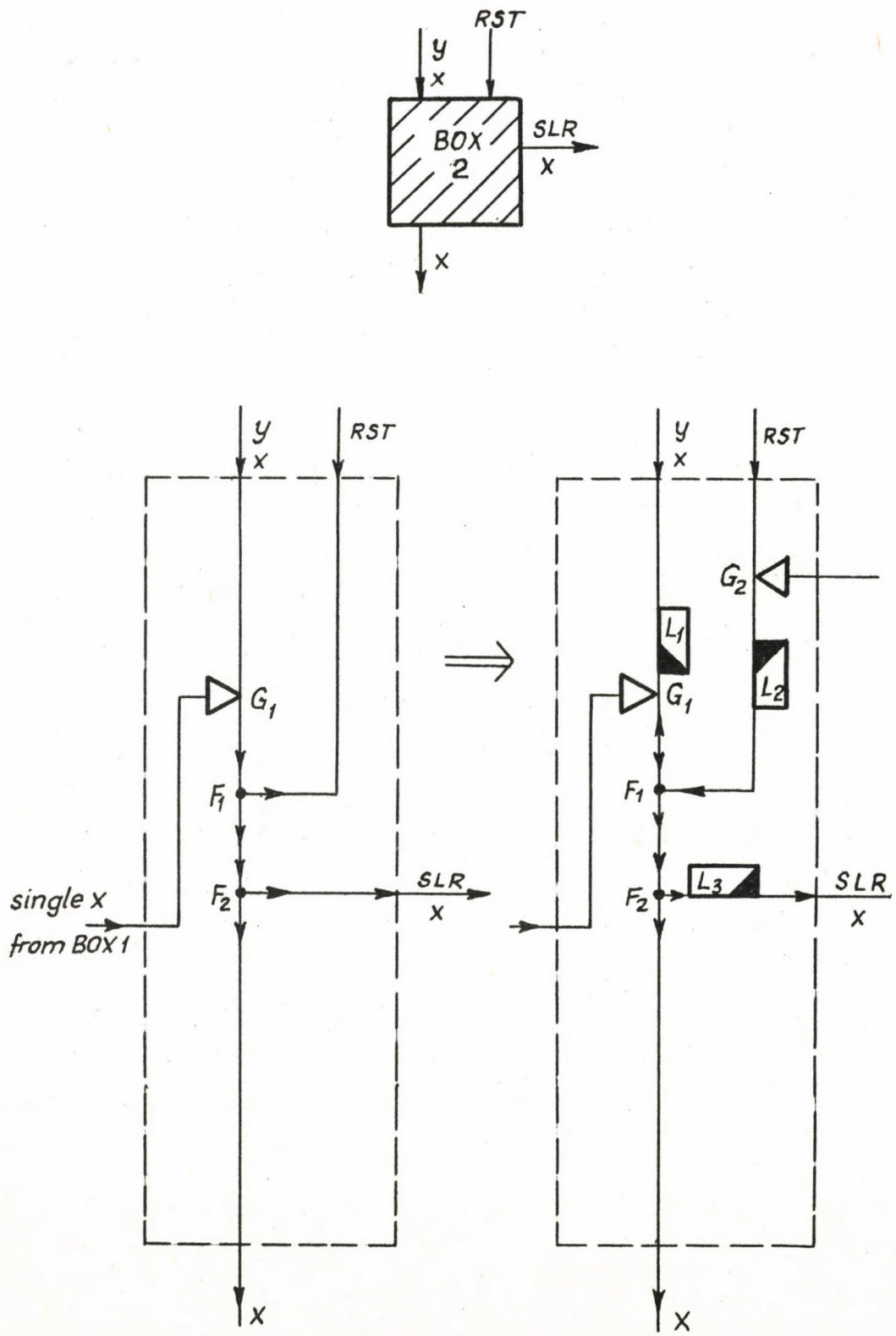


Figure 5.1-5.

Unwrapping BOX 2 of the MAXEL module

Fortunately a clocksignal is automatically provided in the form of the corrector signal propagating along the corrector path. Thus the unwrapping of BOX 3 becomes very simple.

After integrating BOX 1, 2 and 3, bearing in mind that x must not enter the corrector path, and that the lock /lock L_8 on figure 5.2-1/ ensuring this is to be replaced provisionally by a gate to kill the activation signal /gate G_6 on figure 5.2-1/, with new legend one finally gets the principal diagram of the MAXEL module on figure 5.2-1.

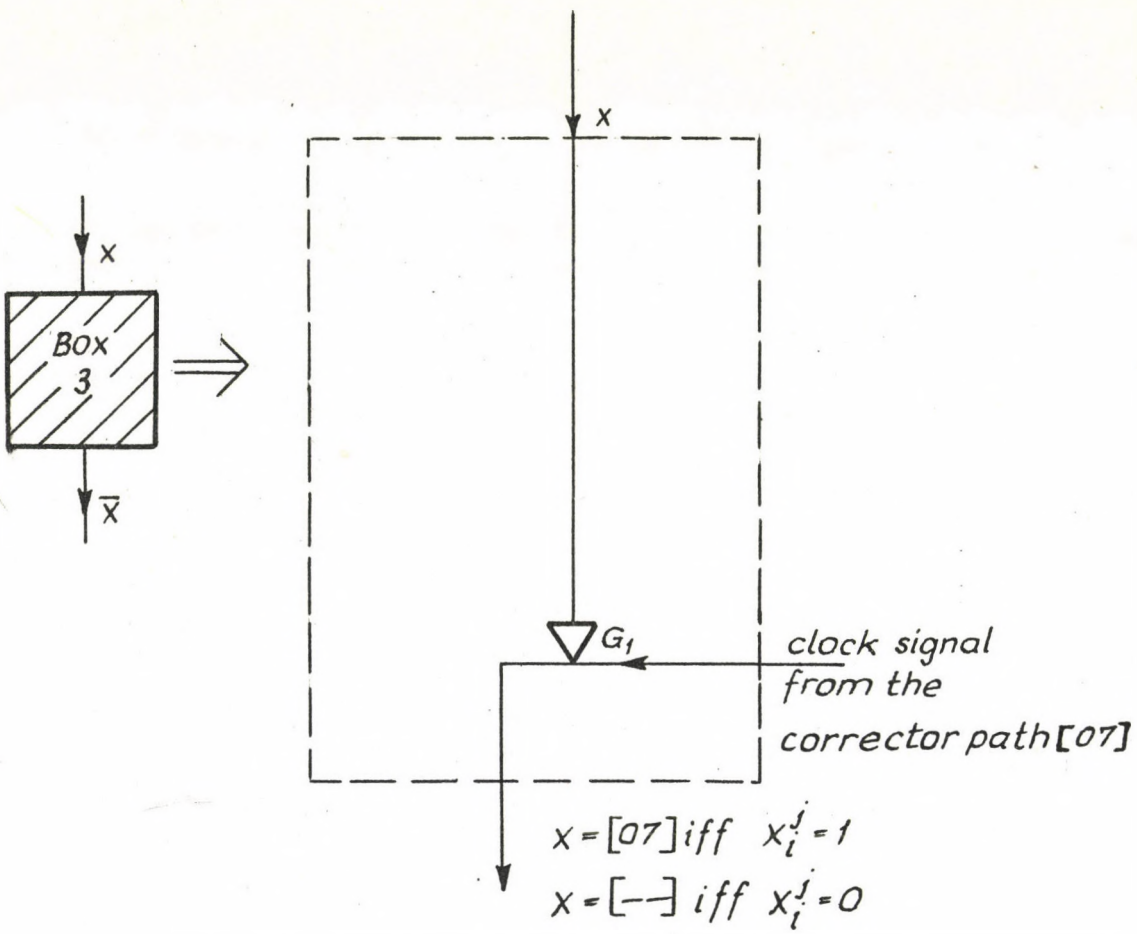


Figure 5.1-6

Unwrapping BOX 3 of the MAXEL module
/Cf. fig. 5.1-2/

5.2 Verbal description of the MAXEL module's operation

Let us consider the final drawing on figure 5.2-1. It refers to the MAXEL module U_i^j, c.f. figure 3.2-1. Suppose, now, that at t = 0 a number n^j, represented by a series of signals [07] or [06] according to $x_i^j = 0$, or 1 respectively, is produced at point E_1.

The only essential property of the signals, representing the information n_i^j, is that the symbol here for the first bit n_i^j is x /instead of the lengthy x_{i+1}^j/ and for the second one y /instead of x_{i+2}^j/, following x by a delay of 26. By a more careful design this delay could have certainly been reduced but some didactical advantages and simplicity would suffer.

At t = 0, as can be seen from figure 5.2-1, each gate is in off state except G_7 which is normally on. Now if at t = 0 x is in cell E_1, then it passes the crossover C_1 and duplicates at F_1. One of the signals goes along path $F_3 \longrightarrow C_4 \longrightarrow C_5 \longrightarrow \longrightarrow C_8$ leaving the unit at point S_1. The second signal /born in F_1/ will propagate along path $F_3 \longrightarrow C_3 \longrightarrow F_4 \longrightarrow F_6 \longrightarrow F_{11} \longrightarrow \longrightarrow F_{10} \longrightarrow C_4 \longrightarrow C_7$. During this, several duplications occur /at F_4-F_6-F_{11} and F_{10}/ but after having transformed into [07] between F_4 and F_6 it reaches $\bar{G}_7$ being in on state. It turns it off before signal x reaches the subordinated path, so it is annihilated. Actually a second transformation [07]$\longrightarrow$[07] takes place owing to the lockpair L_9L_{10} between C_4 and $\bar{G}_7$ but this lockpair has its role only during resetting.

This way the "beheading" of string n_i^j has taken place and all the signals following x, including y, can pass G_7 for it is now in a permanent of state owing to the locking mechanism $F_{10} \longrightarrow F_9 \longrightarrow G_4$. The signal, namely, born in F_9, turnes G_4 on and no other ordinary /information/ signal is able to open it again. Only by the reset signal can it be turned off. See Chapter 10. The pair of the signals, born in F_9, travels along $F_9 \longrightarrow C_3$ but dies at lock L_5 between F_9 and C_3. An other

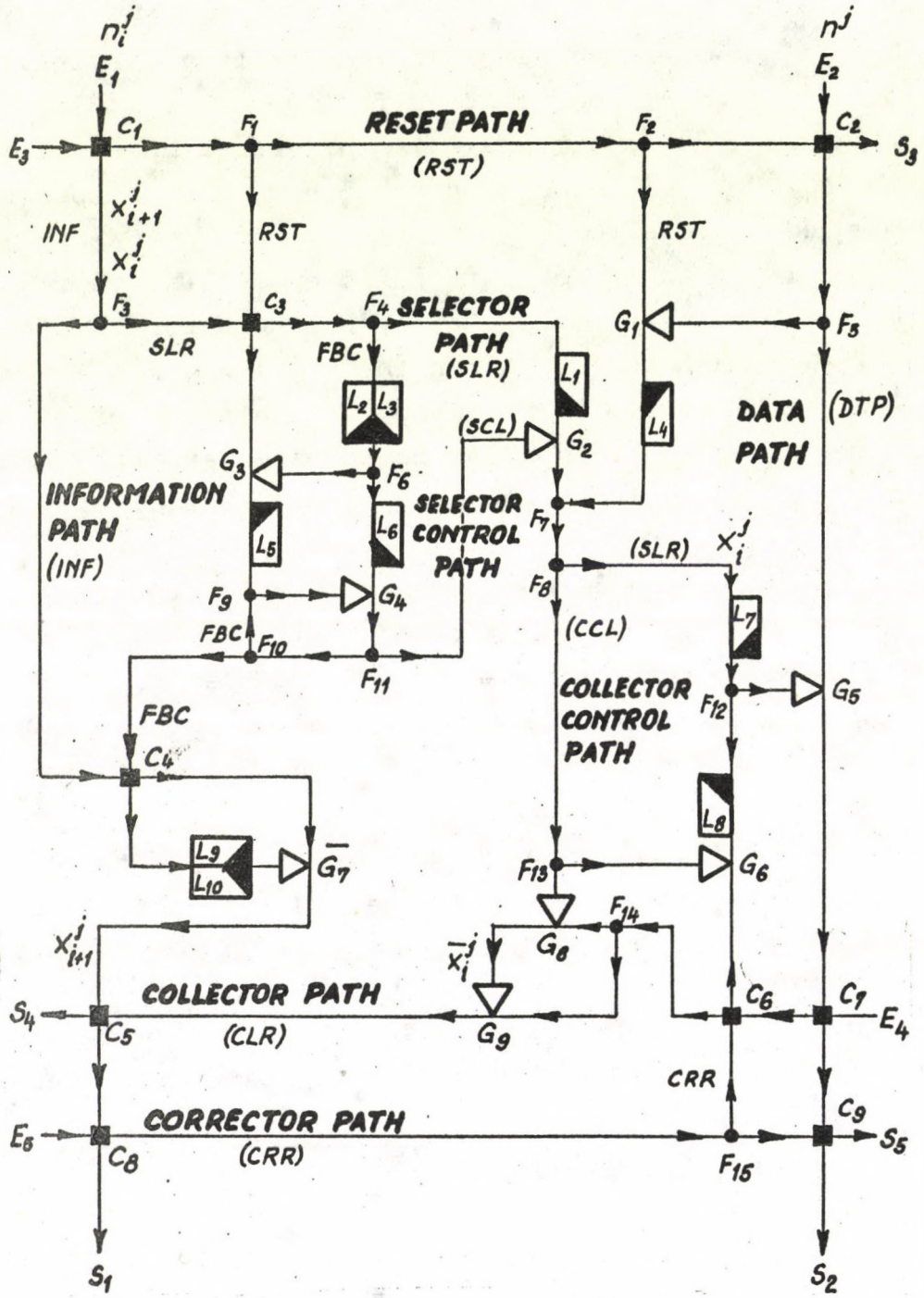


Figure 5.2-1.

Final drawing of the MAXEL module U_i^j

duplication takes place at F_{10} resulting the signals talked over just now. At F_{11} a signal is born again going to turn G_7 off. This turning off, however, occurs *after* the signal x, coming from F_4 to F_7 passes G_2. When the second bit, y, arrives it finds G_2 already on and is annihilated. So are all the signals following y for gate G_2 is now permanently on owing to the lock formed by G_4.

As a result, all but the first bit leave the unit at S_1 and the first bit, but only the first bit, reaches gate G_5 controlling the data path. If $x^j_{i+1} = 0$, i.e. x = [07], then G_5 is on permanently annihilating n^j. If $x^j_{i+1} = 1$, then x = [06], G_5 is off permanently thereby permitting all the bits of n^j to enter the next unit U^j_{i+1} at $/S_2/^j_i = /E_2/^j_{i+1}$.

Meanwhile, signal x is to be inverted to indicate zero. In fact G_9 is on if $x^j_{i+1} = 0$ i.e. x = [07] for x = [07] turns G_8 on, thus inhibiting the collector signal, arriving from $E_4 \longrightarrow C_7 \longrightarrow C_6 \longrightarrow F_{14}$, to turn G_9 on. So G_9 remains neutral permitting finally the collector signal to go to $C_5 \longrightarrow S_4 = /S_4/^j_i = /E_4/^{j-1}_i$.

On the other hand, in case $x^j_{i+1} = 1$, i.e. x = [06], G_8 will *not* be turned on, thus the collector signal, coming from $E_4 \longrightarrow C_7 \longrightarrow C_6$, will turn G_9 on, therefore it will be annihilated. This way the operation is finished and after resetting, the unit is ready for the next operation.

There is a number of components which haven't been spoken of. For instance G_1, G_6, L_4, L_6, C_2 and so on. These are needed only either during activation or resetting but never during normal operation.

Let us stress, here again, that the description above is by no means to be considered as any kind of a *proof* for the correct operation. It is, rather, to help understanding the main processes telling that such and such event triggers such and such events but one does not know, by the description, for sure that there are no other events prohibiting certain processes. Among others, timing is not dealt with in the

necessary detailed way so one cannot state, with certainty that all the propositions mentioned in the verbal description are true.

By Codd and by others computer simulation provided the guarantee for the design correctness. Mathematically, however a simulation can not be accepted for proof just as the running of a program is not the proof of its correctness. For proof correctness there is a promising way of axiomatic approach. Something similar can be done for proving cellular automata design correctness. See Chapters 9 and 10.

Beside operation, activation and resetting are to be cared for.

After staking /i.e. establishing the 0-1 configuration as it is shown in fig.6.3-1/ locks $L_1, L_2, \ldots, L_{10}$ are to be set in, i.e. to bring them in state 3. This can be done by an activating signal a = [07] applied at E_1. It spreads out within the unit setting in all the locks. Each descendant dies, except the one leaving the machine at S_1 owing to gates G_1, G_3, G_6.

As a result, all the gates, apart from now on from the irrelevant gates G_1, G_3 and G_6, as well as gate G_8 to be managed later, but including G_7, will be turned on. By a resetting signal r_1 = [06], however, all the gates will be turned off, finally, by a second resetting signal r_2 = [06] all gates remain off except G_7 which, owing to lockpair L_9L_{10}, receiving the transformed signal r_2^t = [07], will again be turned, and this time, on. Of course, resetting signal should not be sent through the activation path starting with E_1 but rather, through the proper reset path starting with E_3, avoiding the inconvenient t_7-transformer L_2L_3 between F_4 and F_6.

After a normal operation G_7 must be in off state and G_2 must be in on state, owing to the t_7 transformer L_2L_3. The other relevant gates such as G_5, G_8 and G_9 can be equally well in one state out of ON and OFF. Thus a single resetting signal r = [06] applied at E_3 will turn gates G_5 and G_8

certainly off while gate G_7, By the lockpair L_9L_{10}, will be turnded to bring the gates into the necessary initial state.

As for gate G_9 it can be reset from E_4 by a single [06] after having G_8 been reset.

REFERENCES

ALADYEV, V.: Survey of Research in the Theory of Homogenous Structures adn their Applications Mathematical Biosciences, 22, 121-154, /1974/.

BAGYINSZKI, J.: Residue Number System /Decimal/ Arithmetic In Parallel Systems Such As Cellular Automata and Magnetic Bubble Memories KFKI report No.75-15. Hungary /1975/.

BURKS,A.W.: Essays in Cellular Automata University of Illinois Press, Urbana, Ill. /1968/.

CODD,E.F.: Cellular Automata Academic Press, New York and London, /1968/.

CODD,E.F.: A Relational Model of Data for Large Shared Data Banks Comm. ACM 13, 377-387. /1970/.

CODD,E.F.: A Data Base Sublanguage Founded On the Relational Calculus IBM RJ. June /1971/.

CONWAY,J.: Mathematical Games /M.Garder ed./ Sci. Amer. October /1970/.

DETTAI,I.: Effectivity Problems and Suggestions Concerning CODD-ICRA Cellular Space KGM ISzSzI technical report, Hungary /1974/.

DETTAI,I.: A Version of von Neumann's Cell KGM ISzSzI technical report, Hungary /1975/.

DOMÀN,A.: A Three-dimensional Cellular Space /A challenge to CODD-ICRA/ KGM ISzSzI technical report, Hungary /1974/.

DOMÀN,A.: A Flexible Three-dimensional Cellular Space International Congress of Cybernetics and Systems, Bucharest, Rumania, Aug.25-29, /1975/.

DOMÀN,A.: Parallel Computing Systems Doctoral thesis at the Technical University of Budapest, Hungary /in Hungarian/ /1976/.

FAY,G.: Circuitry Realization of Codd's Automaton's Cell First International Conference on Computer Science, Szĕkesfehervar, Hungary, May 21-26.

FAY,G.: DPL ALPHA and Codd's Cellular Space
Second European Meeting on Cybernetics and Systems Research, Vienna, Austria April 19-19. /1974/.

FAY,G.: A Basic Course on Cellular Automata
University lecture notes, Budapest Eotvos Loránd University /In Hungarian/. /1975/.

FAY,G., D.V.TAKACS.:
Survey of de CODD-ICRA
In: LINDENMAYER /1975/.

FAZEKAS,B.: Some New Components and Phenomena in CODD-ICRA Space.
Graduate thesis in mathematics, Budapest Eotvos Lorand University /In Hungarian/ /1975/.

GOLZE,U.: Destruction of a Universal Computer-Constructor in Codd's Cellular Space
Report.Dept. of Comp. and Comm. Sci.Univ. of Michigan, Ann Arbor, Michigan 48104 /1972/.

HERMAN,G.T. /org/:
Conference on Biologically Motivated Automata Theory
MITRE Corporation, Mc.Lean, Virginia, USA June 19-22. /1974/.

HERMAN,G.T. and ROZENBERG,G.:
Developmental Systems and Language
North Holland Publ. Co.Amsterdam. /1975/.

HOARE,C.A.R.: An Axiomatic Basic for Computer Programming
Comm.ACM. 12, 576-583. /1969/.

HUSZAR,A.: Testing equipment for Automaton Cells
Graduate thesis in electrical enigeering.
Technical University of Budapest /In Hungarian/ /1974/

ICRA TEAM.: Studies in Cellular Automata
To be published by Gondolat, Budapest, first in Hungarian later in English. /1976/

IPPOLITO,J.C.:
Contribution a l'implantation de functions logiques sur des reseaux cellulaires
These, Grade de Docteur es Sciences Physiques, Université Paul Sabatier de Toulouse. /1972/

KASZAS,O.: Character Generation in CODD-ICRA Space
Graduate thesis in electrical engineering.
Kando Kalman Technical Highschool, Budapest
/In Hungarian/ /1974/.

LEGENDI,T.: Simulation of Cellular Automata
Inner report, Szeged, Jozsef Attila University
MTA group of Mathematical logic and automata
theory. /In Hungarian/ /1975/.

LINDENMAYER,A./org/:
Conference on Formal Languages, Automata and
Development
Noordwijkerhout, The Netherlands 31 March - 6
April. /1975/.

MARTONI,V.: Investigations in Lindenmayer space.
Inner report KGM ISzSzI Budapest, /In Hungarian/
/1975/.

RIGUET,J./org/:
Conference on Automata and Related Topics
Universite Rene Descartes Sorbonne, Paris
May 16-19. /1974/.

RIGUET,J.: Automates Cellulaires a Bord et Automates
CODD-ICRA I-II.
Comptes rendus de L'Academie les Sciences de
Paris SIRIE A t.282. /19.Janvier 76./ pp.167-170,
239-242. /1976/.

SMITH, III. A.R.:
Cellular Automata Theory
Technical report, Stanford Electronic Laboratories
No. SU-SEL 70-016. /1969/.

SZOKE,M./Mrs/:
Destruction in CODD-ICRA
Graduate thesis in mathematics. Budapest Eotvos
Lorand University /in Hungarian/ /1975/.

TAKACS,D.V./Mrs/:
A Bootstrap for Codd's Cellular Space
First International Conference on Computer
Science,
Szekesfehervar, Hungary, May 21-26. /1973/.

TAKACS,D.V./Mrs/:
Galois Connections and DPL ALPHA System Second
European Meeting on Cybernetics and Systems
Research Vienna, Austria /1974/.

TAKACS,D.V./Mrs/:
Cellular Automaton as a new tool for computer science
KGM ISzSzI report, Budapest, Hungary / In Hungarian/, /1974/

TAKACS,D,V./Mrs/:
Un Automate Cellulaira CODD-ICRA Pour la Recherche de Numbers
These Presente a Universite Rene Descartes de Paris Sorbonne Doctorat d´Universite /1975/.

TOFFOLI, T.: Backward steps in cellular spaces
In: Lindenmayer /1975/.

VON NEUMANN,J.:
The General and Logical Theory of Automata.
pp. 1-41 of Cerebal Mechanismus in Behavior,
The Hixon Symposium, /1948/ ed by L.A.Jeffress,
published by John Wiley, New York, 1951. /1948/

VON NEUMANN,J.:
Theory of Self-reproducing Automata
Universtiy of Illionis Press, Urbana edited and completed by A.W. Burks.

SEMANTIC THEORY FROM A SYSTEMATICAL VIEWPOINT

H.Heiskanen
Helsinki University of Technology
Otaniemi, Finland

A semantic theory based on a mathematical linguistic systems theory is introduced. This theory has been developed for the purposes of content analysis. It presents the principles according to which the meaning of lexical elements can be expresed by means of mathematical concepts such as variables, values of variables, entities, and relations. This semantic theory can also be used for the analysis and comparison of meanings of various lexical elements, and for the description of a vocabulary.

A semantic analysis of a given natural language poses difficulties because of the complexity and apparent vagueness of relevant phenomena. For this reason a precise theory of meaning, i.e. a semantic theory, has not yet been formulated, although several attempts have been made[1]. Similarly, in this article a thought construct will be presented which, with some justification, can be conceived of as a semantic theory.

Much of the work published by semanticists is based on the componential approach. They attempt to describe the structure of vocabulary in terms of the various possible combinations of a relatively small set of very general elements, such as components, markers, or sememes, in a particular language. The linguistic mathematical approach introduced here has the same basis, but a slightly different approach. The principal idea is that the meaning of lexical elements, i.e. the definitions of concepts, can be conceived as formed of values of variables. This means that semantic components are replaced by values of

variables. The other difference from the conventional componential analysis is that the number of semantic components, i.e. values of variables, is not limited; rather it varies with the subject area of the vocabulary and with the familiarity of the audience with this area.

The semantic theory introduced in this article is based on a linguistic mathematical systems theory, or LM-theory. It describes the way in which linguistic concepts can be transformed into mathematical concepts such as values of variables, variables, entities, and relations[3,4,5]. LM-theory was developed for the purposes of analyzing pay claims and pay decisions: a content analysis method based on this LM-theory has been applied for the analysis and design of pay schemes.

THE SYSTEMS THEORETICAL BACKGROUND

The basic overall purpose of semantic analysis is generally seen to be the explanation of how the sentences of a natural language are understood, interpreted, and related to states, processes, and objects in the universe[1]. LM-theory the principles according to which the real world, or alternatively the information given about it through natural language, can be divided into these states, processes, and objects, but LM-theory applies different terms:

- states are interpreted to be values of variables,
- objects to be entities, and
- processes to be either relations, or events.

Variables and their values, entities and relations are the most essential basic concepts of LM-theory.

States. Variables

A variable is formed of two or more mutually exclusive states (or values). These values are, for instance, adjectives such as "*high*" and its antonym "*low*". Together they form a variable "*height*".

LM-theory also includes a symbolic language. By convention the symbol used for a variable is a lower case letter, usually p.

Its superindex refers to the quality of variable, and the subscript to the value of variable.

Example 1

Adjective	Its antonym	Variable	Symbol
long	short	length	p^1
high	low	height	p^2
wide	narrow	width	p^3
heavy	light	weight	p^4

Example 2

p^1 = length (a variable)

p^1_1 = short (a value)

p^1_2 = standard of comparison, which neither is short or long (another value)

p^1_3 = long (third value)

Objects. Entities

An entity is defined to be the thing to which the value of the variable belongs, or which is described by means of this value. For instance, the "*cigarette*" (which is long) is an entity, because the value "*long*" describes its state.

The symbol for an entity is a capital letter, or a combination of letters beginning with a capital letter, usually En. Again the superscript refers to the quality. Thus, for example, En^1 could be "*tower*", En^2 "*cigarette*", and En^3 "*square*". The subscript is used to express the identity of the entity: for instance, En_1^1, En_2^1, and En_3^1 could be three individual towers. This identity signifies which particular object is under examination.

Example 3

En^1 = tower (an entity-category)

En_1^1 = this tower (an entity-individual)

En_2^1 = that tower (another entity-individual).

Processes. Relations

When values, variables, or entities are examined in pairs a coexistence pair is formed. Its members are called partners, and the connection between partners is a relation. These partner-relations may differ from pair to pair with respect to their quality. A relation can be, for instance,

- a *definition-relation*, when one partner is specified by means of the other,
- an *influence-relation*, when one partner influences the other, or
- a *measurement-relation*, when the value of one partner is determined by means of the other.

Example 4

a/ "Pressure depends on temperature".

Here

"*depends*" refers to an influence-relation, and
"*pressure*" and
"*temperature*" are variables.

Thus the content of this sentence is

"variable '*temperature*' influences variable '*pressure*'".

This sentence describes an influence-relation between two variables.

b/ "The pressure is high, because the temperature is high".

Here

"*high pressure*" is a value of variable "*pressure*",
"*high temperature*" is a value of variable "*temperature*", and
"*because*" is a cause-effect-relation.

Thus the content is

"value '*high*' of variable '*temperature*' is the cause and the value '*high*' of variable '*pressure*' is the effect", or

"value '*high temperature*' coexists with value '*high pressure*'".

This sentence is a description of a value-pair, i.e. of two coexisting values.

The relations are symbolized by various strokes and dashes. Thus, for instance, a dash with an explanatory sign "*def*" ($\overset{\text{def}}{=}$) refers to a definition-relation, while a stroke with an arrow head (⟶) to an influence-relation.

Descriptive Sentence. Message

The sentences of a natural language are divided into two categories: descriptive and defining sentences.

The descriptive sentences, which can also be called messages, describe real world states by means of concepts, i.e. lexical elements. The defining sentences or definitions tell what is meant by these concepts. They specify the lexical elements by means of other elements, which this time can be called semantic elements.

Each descriptive sentence can be interpreted to be formed of values, variables, entities, and relations. The analysis of messages is based on this idea, so that the message content is divided into parts corresponding to any of these categories, and then described by means of these parts.

Example 5

"Concept A is a synonym of concept B".

Here

"*concept A*" can be interpreted to be an entity (En)
"*synonym*" is a value of a variable (p_1)

"*connection of concept A with concept B*" is this variable (p), and

"*is*" is an equality-relation (=)

Thus the content of this sentence is

"'*concept A*' has in the variable '*connection with concept B*' the value '*synonym*'", or

"entity En in variable p has the value p_1".

The variable '*connection with concept B*' is formed, for instance, of the following values:

p = connection with concept B

p_1 = synonym

p_2 = antonym

p_3 = homonym

p_4 = hyponym

The kind of sentence introduced in example 5, i.e. a sentence which presents the value of one variable of one entity, is an element-message, because it is the smallest possible sentence. Although more complicated sentences can be formed or deduced from these element-sentences, the way in which this takes place is not relevant to the subject of this article.

Defining Sentence. Definition

In componential analysis of meaning, it is customary to define the meaning of a lexical element in term of semantic components and logical constants[6]. In LM-theory, the same holds, but only the terms are different:

- the lexical element is considered to be a concept,
- the semantic components are considered to be values of variables, and
- the logical constants are considered to be connectors.

In somewhat more detail, this means that each definition can be divided into the following parts:

- the *defined concept* which is the concept specified in the definition,
- the *definition-concepts*, which are the concepts by means of which the defined concept is described. They are
  - the *defining concept*, meaning the concept category to which the defined concept belongs according to the definition, and
  - the *defining variables*, which describe the way in which the defined concept differs from other concepts included within the category of the defining concept, and
- the *definition-relation*, which connects the defined concept with the definition concepts.

Example 6

"Concept A is a synonym of concept B if it is defined by means of all and the same semantical components and of all and the same logical constants as B".

Here

"*synonym*" is interpreted to be the defined concept, i.e. it is specified in this definition,
"*concept*" is the defining concept, i.e. "*synonym*" belongs to the category of "*concepts*",
"*number of semantical components used in the definition*" is the defining variable, and
"*all*" is its value;
"*quality*" of the "*semantical components*" is another defining variable, and

"*same*" is its value;
"*number of logical constants*" is the third defining variable, and
"*all*" is its value;
"*quality of logical constants*" is the fourth defining variable, and
"*same*" is its value; and
"*if*" refers to the definition-relation.

Thus the content of this definition is:

"'*Synonym*' belongs to

the category of '*concepts*' and differs from other concepts in being defined by

'*all*' (= variable 1) and
'*the same semantical components as another concept*' (= variable 2), and by
'*all*' (= variable 3), and
'*the same logical constants as this other concept*' (= variable 4)".

METHOD

The LM-concept analysis method is based on the idea about the structure of definitons presented above. This means that when this method is employed the definition is divided into parts corresponding to the defined and definition concepts, and definition-relation, and then presented by means of these.

Example 7

"Concept A is a hyponym of concept B, if its definition contains all the same components and logical constants as those occurring in the definition of B in addition to at least one other component with the necessary logical constants".

Interpretation:

Concepts:

"*hyponym*" = defined concept Co^0
"*concept*" = defining concept Co^1
"*number of semantical components compared with those in the definition of B*" = defining variable p^1
"*all+at least one*" = value p^1_1

"*quality of the components compared with those in the definition of B*" = variable p^2
"*same*" = value p_1^2
"*number of logical constants compared with those in the definition of B*" = variable p^3
"*all+at least one*" = value p_1^3
"*quality of the constants compared with those in the definition of B*" = variable p^4
"*same*" = value p_1^4
"*if*" = definition-relation $\underline{\text{def}}$

Content:
"Co^o belongs to the category of Co^1 and has in variable p^1 value p_1^1, in variable p^2 value p_1^2, etc."

Comparison of Definitions

When each individual definition has been analyzed as described above they can be compared with each other. This is accomplished in the form of a "*substraction*". The analyzed definitions are written one above the other and then compared with each other with respect to each definition concept.

Example 8

Table I presents the comparison process schematically. On the left are the definitions to be analyzed and compared. They are the "*synonym*" from example 6, the "*hyponym*" from example 7, and the "*antonym*". Each definition concept is allotted a column and the results of the analysis of these definitions are entered in the appropriate columns.

The notations in each column are then compared with each other. Thus the three example concepts are similar with respect to the defining concept, i.e. all of them are "*concepts*", and to the fourth defining variable, i.e. they are defined by means of the "*same*" logical constants than

another concept. But they are nonsimilar with regard to the first, second, and third defining variable, in which they have different values.

DESCRIPTION OF A VOCABULARY

Every vocabulary can be conceived to be a set of concepts in which the concepts are tied together by means of definitions. This means that some of its concepts are defined by means of the other.

According to the LM-way of thinking, the structure of such a set, i.e. the interdependencies of its lexical elements, can be described in a two-dimensional scheme.

In the horizontal direction the description takes the form of a network where the concepts are joined by means of definitions. Thus in this network the lexical elements are the knots or nodes, and the definitions the meshes (figure 1).

In the vertical direction the concepts are described by means of the concept hierarchies.

The Horizontal Description

The horizontal description of a vocabulary begirs by analyzing some of its central concepts in the way illustrated in example 7. This means that the central lexical element will be spread out into defined and definition concepts connected by the definition-relation.

Example 9

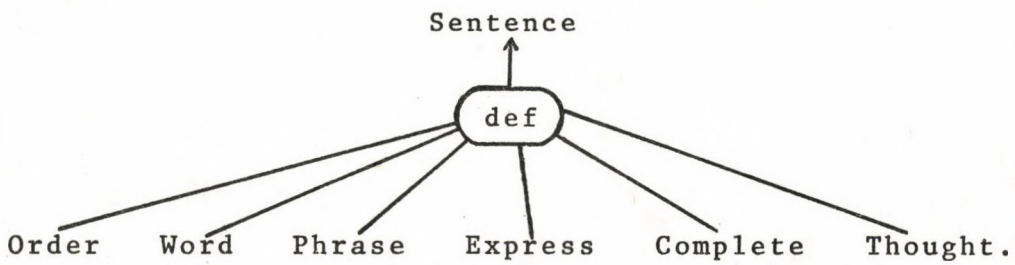
"Sentence is an order of words or phrases expressing a complete thought"[7].

Interpretation:

Concepts:

- defined concept = " *sentence*"
- definiton concepts = 1/ "*order*", 2/ "*word*", 3/ "*phrase*", 4/ "*express*", 5/ "*complete*" and 6/ "*thought*"
- definition-relation = "*is*"

Content:



In the next step which is presented in example 10, the definitions of the definition-concepts of the original sentence are analyzed in a similar way. This time they are in the role of the defined concept.

Example 10

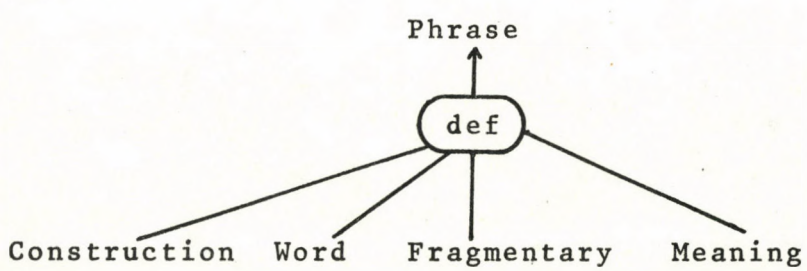
"Phrase is a construction of words having fragmentary meaning".[7]

Interpretation:

Concepts:

- defined concept = "*phrase*"
- definition-concepts = 1/ "*construction*", 2/ "*word*", 3/ "*fragmentary*", and 4/ "*meaning*"

Content:



So this procedure goes on: the definitions of the definition-concepts of the former step will be analyzed and spread out into above described networks. This is continued until the basic or *undefinable* concepts have been found. This takes place when two concepts have been found to be defined by means of each other.

Example 11

"Expression is utterance".[7]

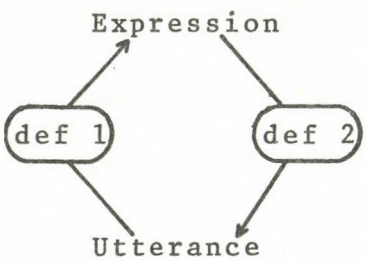
"Utterance is expression".[7]

Here

"*expression*" the defined concept in the first definition, and the definition concept in the latter, and

"*utterance*" is the definition concept in the former and the defined concept in the latter.

Thus the content is:



The "*expression*" and "*utterance*" are specified by means of each other. This means that they are undefinable concepts which the definer thinks to be comprehensible to the audience without definitions.

When all the definition chains have ended at undefinable concepts, the whole vocabulary can be presented in the form of a summary, in which the formal descriptions of the analyzed definitions have been combined into one and the same network.

Example 12

In figure 1 a network of a vocabulary is introduced. It describes the vocabulary used in Webster's dictionary[7] in defining the word "*sentence*".

This network starts from the definition of "*sentence*" analyzed in example 9. This definition is denoted by "*def 1.1*". - The numeral to the left of the decimal refers to the level of analysis, and the numeral on the right to the order of the definitions on this level. Thus, for instance, 2.3 means that it is the third definition of the second level.

On the second level are the definitions of those concepts by means of which "*sentence*" has been defined. So, for instance, the definition of "*phrase*" analyzed in example 10 is the definition 2.2, and the definition of "*expression*" is 2.3.

On the third level the definition concepts of the second level are analyzed. Here are, for instance, the definitions of "*neat*" (3.1) and "*arrangement*" (3.2). - On this level some undefinable concepts have been emerged, such as "*utterance*" (3.6) and "*entire*" (3.7). These undefinable concepts are in the network underlined and provided with an ordinal number.

Figure 1 expresses six levels of analysis. However, at the sixth level some concepts remain which are not undefinables. These include concepts such as "*assortment*", "*miscellaneous*", and "*ingredient*" (6.9). But their definitions will be left out the network due to the space restrictions.

The network description of a vocabulary encompasses two kinds of lists. The first is comprised of the definitions of the terms, and the other of the undefinable concepts used in these definitions.

Example 13

Here is the list of the undefinable concepts of the network in figure 1.

1 utterance (expression)
2 entire (complete, not fragmentary)
3 idea (thought)
4 proper (fit,neat)
5 put (to place)
6 union (combination)
7 fraction (part)
8 take (receive)
....

The Vertical Description of a Vocabulary

Each concept in the horizontal a network is an element of a concept hierarchy. The description of these hierarchies is the vertical description.

A concept hierarchy is formed when a concept is divided into two or more mutually exclusive subconcepts, and these again further into new subconcepts. At the top of such a hierarchy are the most abstract concepts, and at the foot the most concrete ones. - Essential in this division into subconcepts is that it takes place by means of definitions. Each concept which will be divided, is provided with one or more defining variables by means of which the subconcepts are then specified. Thus

- the original concept is in the role of the defining concept, and
- its subconcepts in the role of defined concepts.

Example 14

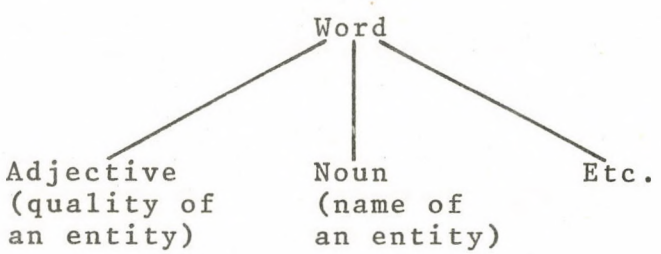
In figure 1 the term "*word*" is included in definitions 1.1, 2.2, and 3.4. The conceptual hierarchy of this term will be derived in the following way:

"'*Words*' are divided into nouns, adjectives, pronouns, verbs, etc."
"'*A noun*' is the name of an entity".
"'*An adjective*' belongs to the quality of the subject".
"Etc".

In these definitions

"*word*" is the superconcept,
"*nouns*", "*adjectives*", "*pronouns*", "*verbs*" etc. are its subconcepts
"*type of concept to which the words belongs*" is the defining variable, and
"*name of an entity*" and
"*quality of an entity*" are its values.

Thus the content is:



In example 14 it has been shown in what way concepts are divided into subconcepts. The basic idea is that is is accomplished by means of definitions and that the defining variables serve as the criteria of the division.

The subconcepts will be divided into further subconcepts by means of new defining variables. When these definitions will be combined into one and the same network a hierarchy will be formed. A detailed description of such an hierarchy comprises on the other hand of a description in what way the concepts are organized into super- and subconcepts, and on the other hand of the defining variables used as criteria of division.

Example 15

In figure 2 such a concept hierarchy is presented. On the left are the concepts, organized into a hierarchy; on the right are the defining variables used in this organization, and their values. Thus for instance, the nouns or "*entity-names*" can be divided into nouns describing "*abstract*" and "*concrete*" entities. Here the defining variable is the "*concreteness*" and its values are "*abstract*" and "*concrete*". - The concrete entities can be divided into "*immaterial*" and "*material*" entities. The defining variable is "*materiality*" having the value "*immaterial*" and "*material*".

Typical for the hierarchies of real vocabularies is that there are words, or names for each concept in the hierarchy. Normally, those concepts which are used seldom do not have a name, but must be described by means of one or more adjectives and another noun. In figure 2, for "*instance*" "*other animals than men*" or "*not living objects*" do not have a noun of their own, and they must be described by means of adjectives.

Characteristics of a Vocabulary

The hierarchic structure of a vocabulary is one essential dimension which describes various vocabularies. A vocabulary which has very high hierarchies, and which has also a special name for each of the concepts in these hierarhies is a noun-dominated vocabulary, where as a vocabulary with low hierarhies is an adjective-dominated one. In the former there is an enumerous amount of nouns and few adjectives, where as the latter has few nouns, but many adjectives. The extreme cases of these are on the one hand a vocabulary with only nouns and with no adjectives, and on the other hand a vocabulary with many adjectives and one noun (which is entity).

Example 16

The term "*widower*" in figure 2 is an expression of a noun-dominated vocabulary. It could be expressed by means of a extreme adjective-vocabulary in the following way:

"A '*widower*' is an

. entity, which is
. concrete,
. material,
. living,
. able to move,
. two legged,
. male,
. married, and
. has lost his wife through death."

A noun-vocabulary has an advantage of enabling the formulation of short sentences, but its drawback is the width of its vocabulary. An adjective vocabulary is just the opposite: it has much smaller vocabulary, because by a rather small amount of adjectives can be replaced a much larger amount of nouns, but the sentences formulated by it are very long, because every noun may be described by means of several adjectives. Thus either the vocabulary will widen out, or the sentences will lengthen depending whether the number of nouns is increased or decreased. Normally the practical vocabularies are approximately in the middle: they consist of a rather large number of nouns and adjectives: those terms used often are provided with nouns and the rare concepts are described by means of adjectives and other nouns.

APPLICATIONS

The LM-theory, and the content analysis method based on it, can be used for the purposes of semantic studies as in the analysis of meanings and descripition of vocabulary as discussed above. It can also be applied in other scientific areas to analyze concept definitions and to describe concept sets used in theoretical schemes. So, for instance, it has

been used for the analysis of wage theories and theories of action in order to compare and combine them into comprehensive theory.

Discussion

In the introduction of this article, the linguistic mathematical approach or LM-theory has been said to be a semantic theory. This assertation is based on the definition of semantic theory presented by Bierwisch[1]. According to him it must

- make reference to the syntactic structure in a precise way,
- systematically represent the meaning of lexical elements, and
- show how the structure of the meaning of words and the syntactic relations interact in order to constitute the interpretation of sentences.

In this article I have tried to prove that these conditions are fulfilled by the LM-theory at least to some extent.

The LM-theory includes a syntactic theory, though it has not been discussed per se in detail in this article. However, examples 4 and 5 suggest on the one hand, how this syntactic theory is involved, and on the other, that a precise connection exists between the semantic and syntactic structures. Thus the first requirement should be met.

This entire article has been directed at introducing LM-theory as a systematic method for presenting the meaning of words, or concepts. Thus also the second condition should be fulfilled.

The LM-analysis method shows the way in which the sentences can be systematically interpreted, i.e. first, divided into concepts, and then, presented by means of these concepts as introduced in examples 4 and 5. The concept definitions are

then analyzed in a similar way: first, divided into concepts, then, presented by means of these. This should show the way in which the meanings of words and the syntactic relations in teract. Thus also the third condition has been met.

Based on this reasoning I feel justified to some extent in asserting that LM-theory is a semantic theory. One could also add the adjecitve "*practicable*", because it has been applied to such practical and conrete problems as the analysis of pay decisions in day-to-day working situations.

References

[1] Bierwisch,M.:"Semantics" in New Horizons in Linguistics (ed.J.Lyons), Penquin Books, Middlesex, 1972.

[2] Carnap,R.: Meaning and Necessity, University of Chicago Press, Chicago, 1956.

[3] Heiskanen,H.: "Palkan muodostumisen teoriaa ja käytäntöä" ("Theory and practice of wage formation) Helsinki University of Technology, Research Papers No.38., Otaniemi, Finland, 1972.

[4] Heiskanen,H.: "A Mathematical Linguistic Theory", Helsinki University of Technology - Laboratories of Industrial Economics, Report 5/1974, Otaniemi, Finland

[5] Heiskanen,H.: "A Systems Theoretical Approach to the Analysis of Social Action and Decisions", Annales Academiae Scientiarum Fennicae, Series B No 191, Helsinki 1975.

[6] Katz,J. and Fodor,J.: "The Structure of a Semantic Theory", Language 39:1963:170-210.

[7] The Grosset Webster Dictionary, New York, 1974.

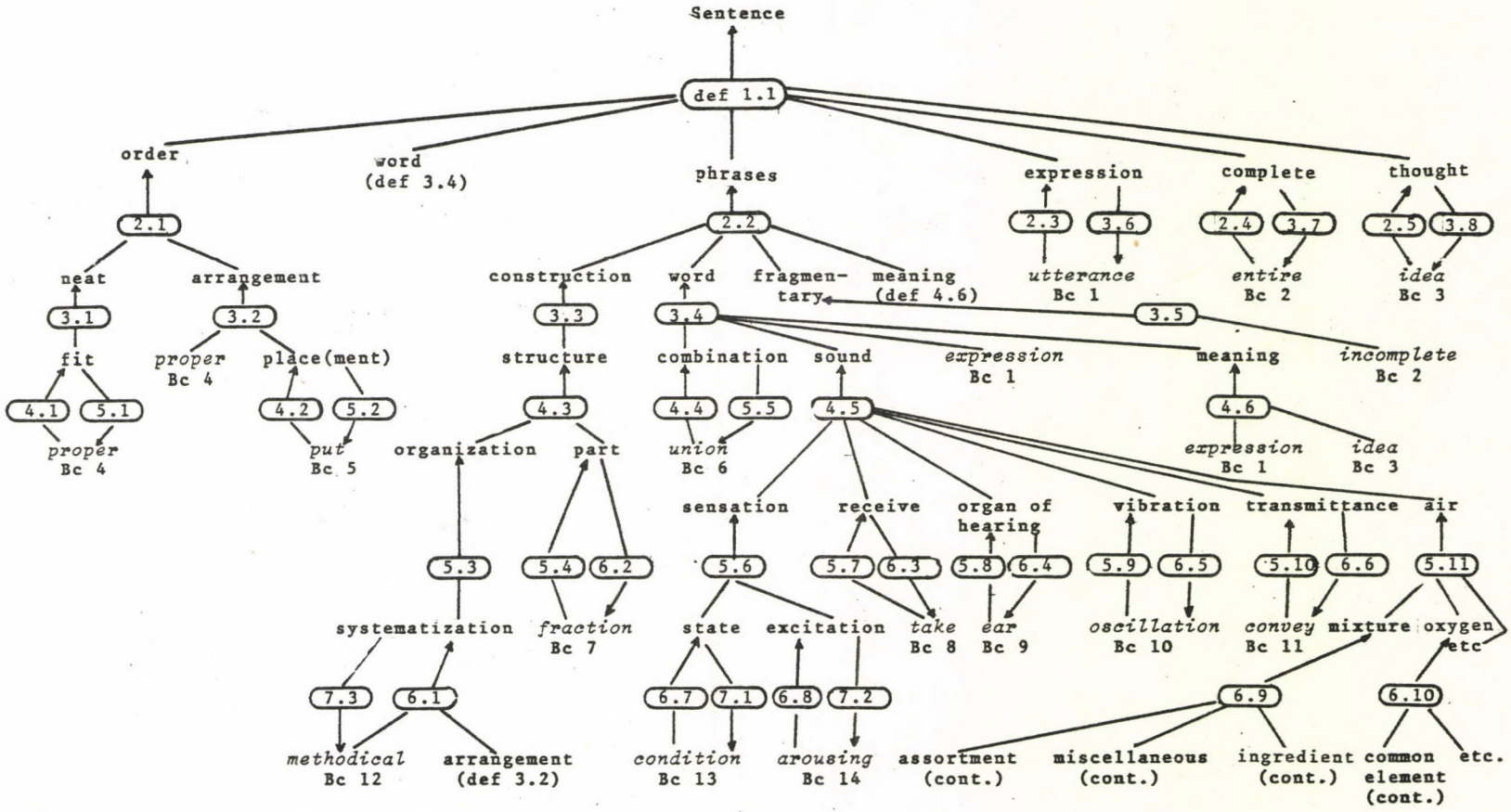
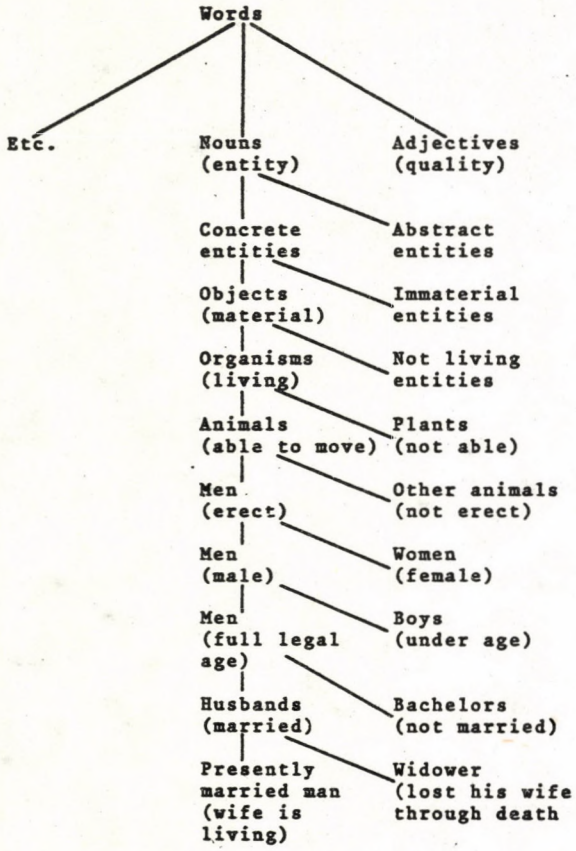


Figure 1. *Vocabulary as a network of definitions*



Definition: "Word is an articulate sound .... expressing an idea"

Defining variables

Variable	*Value*	*Value*
Type of idea for which the word is symbol	Subject or entity	Quality of the subject
Conreteness of the entity	Concrete	Abstract
Materiality of the entity	Material	Immaterial
Livingness	Living	Not living
Ability to move	Able to move	Not able
Position	Erect	Not erect
Sec	Male	Female
Age	Under age	Full legal age
Marital status	Married	Not married
Livingness of his wife	Living	Not living

Figure 2. *Concept hierarchy*

Analyzed concepts and their definitions Co^o	Defining concept Co^1	Defining variables				
		Number of semantical components compared with those of another concept p^1	Quality of semantical components compared with those of another concept p^2	Quality of logical constans compared with those of another concept p^3	Quality of logical constans compared with those of another concept p^4	Type of difference of the differing semantical components p^5
Concept A is a *synonym* of concept B, if it is defined by means of all and the same semantical components and of all and the same logical constants than B. $= Co_1^o$	concept Co_1^1	same p_1^1	same p_1^2	same p_1^3	same p_1^4	(not relevant)
Concept A is a *hyponym* of concept B, if its definition contains all the same components and logical constants in connection with these as occurring in the defintion of B an in addition at least one other component with necessary logical constants. $= Co_2^o$	concept Co_1^1	same+at least one p_2^1	same p_1^2	same+at least one p_2^3	same p_1^4	(not relevant)
Concept A is a *antonym* of concept B, if its meaning is identical with the meaning of B, except that its meaning has a component C1 that of B had C2 and C1 and C2 belong to particular subset of mutually exclusive components $= Co_3^o$	concept Co_1^1	same p_1^1	same except one p_2^2	same p_1^3	same p_1^4	mutually exclusive p_1^5
Comparison	=	≠	≠	≠	=	≠

Table 1. Comparison of concept definitions

CALLPROCESSORS IN COMPUTER ARCHITECTURE

T.Legendi
Attila József University
Szeged, Hungary

The real effectivity of digital computers - characterized by the working/waiting ratio of basic elements, gates and bits - *is very low*. There are speed limits for the basically sequentially organized computers too.

Cellular automata organization offers *in principle solution* to these problems. However *traditional approach* - stand alone cellular computers consisting of a high number of cells with a fixed relatively big transition function - give results of only *theoretical importance* at the present state of technology - which demands at the same time the development of a homogeneous basic element.

This paper gives proposals for the structure and programming of *very effective medium speed cellprocessors based on existing technology:*

- *Callprocessors* are trested as an *organic part of computer architecture* for solving tasks of a wide but limited class of algorithms.

- The *flexibility* of cells is increased by *variable transition functions* which *reduce the size of cells* and the number of cells for solving a given task.

- Reorganization of processing in a cellular space - *centralized sequential transition function processing* -

drastically reduces the size of cells thus making possible to integrate 10^2-10^3 cells on one chip. With the growing number of cells executing the same transition function *price/performance is improving*, since the loss of speed depends only on the size of the transition function. /This advantage against supposed parallel spaces is independent of the development of technology, too./

- A 16 state cell for general computations and a 2 state cell for specific applications have been designed and their models have been built.

- Cellular automata cross-software - simulation languages and a transition function minimization language - have been implemented.

- General principles of cellular space programming - mapping algorithms on sets of interrelated processing elements connected and working in pipe-lines - have been introduced. The conception of a cellular macro-assembly language and directions towards higher level cellular languages are shown.

1. THE ACTUALITY OF THE TOPIC AND THE GOALS OF RESEARCH

The object is justified by the low level of effectivity of computers in common use; 10^{-4}-10^{-6} part of the hardware components work useful at the same time /although principally all of them could work in parallel/. Speed limit for sequential processing also involves other organization principles.

Cellular automata could satisfy in principle these requirements.

The implementation of cellular automata for practical purposes first become realistic with the advent of the LSI technology. The *heterogeneity of LSI* circuits /because of their complexity/ means at the same time *demand* for a *homogeneous basic element* suitable for mass production.

According to estimations [13] further development of technology and detailed research will generalize the commercial use of homogeneous computers, but not before the mid 80'-s.

This paper reports our research in order to speed up this process. The main *goal* is to design a *smaller new basic element* built on the basis of *existing technology* and to ensure *more effective and/or faster processing*.

2. THE ROLE OF CELLULAR AUTOMATA IN COMPUTER ARCHITECTURE

Theoretical [1, 2] and practical [13,3,4] works concentrate mainly on the idea of a *stand alone cellular automata-computer based on cells with fixed transition function.*

The main advantage of this approach is the total homogeneity /of neighbourhood and transition functions/ which simplifies theory, hardware design and programming of the cellular space.

The relatively small programmable basic element /cell/ guarantees deep simulation level /gates, hardware constructions may be embedded into cellular spaces by software means/.

However there are arguments against a stand alone cellular automata-computer. The most important of them states that *cellular automata* /especially taking into consideration the existing technology/ *cannot be applied in an economical way for the execution of arbitrary tasks.* The main reason for this is that in general it is very hard to utilize the level of parallelism of cellular spaces.

Among other problems the initialization of the space and I/O in the traditional way /only through dummy cells/ are quite inconvenient and slow.

The connection of a cellular computer to mass storage is also unsolved.

It is obvious to use digital computers to solve the above problems. For example a computer can load the initial configuration /program/ into the cellular space and can handle the I/O too. In this way there is no need for large and slow configurations /the own software of the cellular automata/, since computers may assemble cellular programs too.

Special purpose applications may be satisfied by a system consisting of one digital processor and one cellular processor only where the cellular automata may be interpreted as a peripherial firmware which is able to execute an extra instruction of the digital computer /see.e.g. [6] where picture preprocessing is executed speeding up the computation with a

factor of 10^4/.

Another possible construction would be to apply a computer as a front-end processor to a larger cellular space.

For *general* computations it is reasonable to use *cellular automata as processors in an architecture* where they execute algorithms effectively computable in parallel way at cell level. Examples are indicated in the chapter on cellular programming. Extending the class of effectively computable /in cellular spaces/ algorithms is an important research task. Algorithms outside of this class should be executed by other processors in the architecture.

3. THE TRANSITION FUNCTION PROBLEM

A fixed transition function is not flexible enough and as a consequence it must be relatively large to be universal. In this way the use of *variable transition functions* also helps considerably to decrease the cell size. A fixed transition function may be interpreted as a union of partly defined functions /subfunctions/ [1,2,5], since practically, in different groups of cells /only/ different subfunctions are used /rather than the whole transition function/ during longer periods. In this way a space consisting of *groups of cells with independently variable functions can replace a space of cells with a larger fixed transition function.*

There is another obvious advantage - the possible use of arbitrary functions /not only a fixed set of subfunctions may be accessed/.

The definition of transition functions for different computations is a cell microprogramming task which is to be made now by hand /in the future this work should be automated by cellular programming languages first partly, then totally/.

Cell microprogramming means a big economy in configuration /program/ size, thus in execution speed too; using special microprograms small groups of cells or even individual cells may replace larger configurations of a space with a fixed transition function.

Cell microprogramming does not exclude production homogeneity.

Assembling and loading of microprograms for cellprocessors is the task of other processors in the architecture.

4. CELLPROCESSORS

The size /and therefore the economy and programmability/ of cellprocessors is determined mainly by the size of the basic cell. A fixed set of states and neighbours always reflects a compromise: in a space of smaller cells more cells are needed for a given task /space, speed/; in case of bigger cells less cells are needed, speed is growing, but the cells are less utilized [5].

Theoretical and simulation results as well as the existing technology suggest to choose for basic element a maximum 16 state cell with a /static/ neighbourhood of 4 to 8 cells. For special classes of algorithms 2 state cells may offer special advantages.

In the previous chapter the necessity and advantages of the variable transition function were explained. Here we emphasize that its use reduces the size of the basic cell and the number of cells needed for a given task.

The technical solution does not involve any serious problem. Existing design methods and technology determine the use of RAM memory for storing transition functions.

In this case the next state of a cell is defined by mapping the state word composed of the neighbour cells - including the cell itself - onto a contiguous address interval of the RAM where the result points to the value of the next state. This is a totally homogeneous construction from the production point of view.

However a cell containing a RAM represented transition function /even with relatively few states and neighbours and some limitations for possible transition functions/ is quite expensive as compared to the computational power of a single cell.

A relatively obvious method may be suggested to improve the price/performance ratio. It is based on the contradiction between the sizes of the memory and the processor part of a cell. A cell, as a basic element of a typical distributed computing system, has a memory component /its own state, 1-4

bits/ and an information processing component /finite automata with 4-20 bits of input and 1-4 bits of output/. It is very natural *to centralize the automata part* of cells having the same transition function. In this case, a cell does not include any processor component. It has only a memory component plus a small additional circuit which interacts with the neighbour cells and with the centralized transition function and stores the resulting new state in the memory component.

This organization results in slower speed, cheaper, modular construction. Such a quasi-space may be interpreted as a cellular space emulator taking into consideration the usual definition. This quasi-space emulator works as an ordinary space emulator - only at a lower speed.

This centralization is naturally modular in the case of RAM stored transition functions and eliminates a specific problem here: namely loading the transition function into each cell may cause problems.

The main advantage of the method is that *the size of a cell is drastically reduced*, giving chances for implementation of 10^2-10^4 cells on one chip which is the *preliminary design condition of an acceptable cellprocessor*, containing 10^5-10^7 cells.

But how price/performance may be improved? The loss of speed seems to compensate the decrease in price. Although the loss of speed is necessary but the solution proposed by T.Toffoli [7] for experimental cellular space emulators makes *proportional the loss of speed to the length of the microprogram* /proportional to the number of terms of the transition function/ *rather than to the number of cells* connected to a RAM stored microprogram. This means that increasing the number of cells belonging to the same RAM, price/performance is improved. This argumentation shows not only that in this way there are chances for implementation based on existing technologies at present, but it also should be pointed out, that although having better technologies in the future, which will enable to integrate a minimal sufficient number of /non quasi/ cells on one chip they will not make the above organization

unnecessary. Of course, the number of quasi-cells on one chip increases too, and therefore the advantage in price/performance remains. When extra speed is necessary, however, a really parallel space should be used.

Returning to the actual situation the loss of speed /relative to a completely parallel space/ does not effect the priority in speed against sequential or mainly sequential computers /estimated order of 2 magnitudes/ and the advantage in more frequent use of components /estimated order of 3 magnitudes/. The legitimacy of these estimations depends on the concrete parameters of the cellprocessors to be implemented and on the cellular programming and microprogramming.

Two types of quasi-cells with fixed 5 neighbours /including the cell itself/ have been designed: a 2 state one /about 10 gates and in average 8 substeps e.g. 8 microinstructions/ and a 16 state one /about 100 gates, the number of substeps largely depends on the transition function; it may vary from 40 to 1000/.

These results ensure minimum preconditions for producing cellprocessors. Details of the system design may be found in L7.

According to L7 a *basic logical modul* is a *group of cells with a common variable transition* function which may be realized as a *set of physical moduls - cell microprocessors /CMP/ consisting of an internal array of quasi-cells, driven by a common external RAM stored microprogram.*

L7 contains further optimization techniques to reduce the number and time of substeps preserving the relative simplicity of quasi-cells at the same time. Minimization is also supported by software tools /L6/.

According to L7 a *cellprocessor* consists of a set of CMP-s and RAM-s, and a /usual/ microprocessor /with a special instruction set/ which controls the CMP-s and RAM-s including the I/O among them (internal I/O) and the external I/O with the other processors in the architecture.

The programs of a cellprocessor /the programs for the

microprocessor, the cellular programs and microprograms/ should usually be computed /in the optimal case by a compiler of a cellular language/ and transformed by other processors in the architecture.

5. SOFTWARE TOOLS

In order to maintain design and later use of the described cellprocessor system, software tools /running on digital computers/ are requested for microprogramming, machine code and higher level cellular programming.

Three implemented subsystems of a future cross-software package and further design principles are explained in the next paragraphs.

5a The CELLAS cellular space simulation language

In rich literature on cellular automata there are relatively from simulation languages [2,9,10,11,12] and their structures are not suitable for our purposes. In this way it was necessary to define a relatively simple language, considering that a limited inhomogeneity is needed and simulation should be as much effective as only possible. The CELLAS language may be used for more general tasks /then the explicit goals of the author/ but it is far from being a general cellular space simulation language.

For special purpose simulations we have adapted other simulators as the interpreter of the SICELA language [9] /with some improvement,a higher level input language was implemented/ and the CELIA [11,12] processor.

CELLAS is a command-type language implemented in form of interpreters.

The basic group of instructions in CELLAS ensures the direct simulation of the space /input vectors to dummy cells on the boundary of the space may be specified if desired/. The interpreter computes the transition function only on open cells that may change their states; on closed cells /that certainly will not change their states/ it does not perform any operations. A one step look ahead algorithm classifies conti-

nuously the cells open or closed. On permanently /more than 1 step/ closed cells the look ahead does not require any computation. This look ahead algorithm ensures significant increase of speed which is especially needed, since the characteristics of the task demand uneffective simulation /on a relatively big sequential digital computer/ of many relatively small elements working in parallel way.

The group of instructions for *functional simulation* helps in speeding up simulation too. These instructions ensure direct simulation of groups of cells /not cell by cell/ and the connections among groups and cells. The main advantage of this group of instructions is given mainly by the possibility of *top-down programming* rather than by economy. The whole configuration /program/ should be decomposed; the decomposition can be tested by functional simulation and afterwards the parts may be decomposed again or changed to real cell configurations continuously and independently. At each level the space may be simulated, giving a very strong debugging tool at the same time.

Transition function definition instructions enable simple function description including the inhomogeneous case.

From the users' point of view the simple, flexible group of instructions for *control of printing* the space is very important. It is possible to define independently, which parts of the space, in what form and at which steps should be printed /or sent to a file/.

The space may be continuously monitored by ON instructions - when a condition of a previously executed ON instruction is met, the prescribed action /a CELLAS program/ of the same ON instruction is executed. There are in the language *assignment*, very simple *arithmetic*. *I/O* and *library handling* instructions for the basic date types /integers, vectors, configurations, transitions functions and CELLAS programs/. Branching is performed by simple skip instructions.

More detailed descriptions of the language may be found in L4, L2, L3. Different versions have been implemented in

FORTRAN IV on computers CDC 3300, IBM 360/40, Honeywell 6060.

5b The INTERCELLAS, an interactive cellular space simulation language

The language is a subset of CELLAS, except a few added instructions for handling interrupts and for dialogue L5, L9.

A typical simplification is shown by the changed ON instruction group: here the conditions cannot be complex, the action is to pass control to the main input periphery /consol, in general/, i.e. the interpreter prints the condition that was met and waits for the next instruction from the main periphery. Different versions of the language have been implemented on minicomputers CII-10010, PDP-8 and MITRA 15.

5c The TRANSCELL cellular microprogramming language (for definition and minimization of the transition function)

The size of a microprogram effects RAM costs, but this is not too serious taking into consideration the relatively great number of quasi-cells belonging to the same RAM.

However, the size of microprograms is critical, since the speed is proportional to the length of the microprogram as it is executed sequentally.

Therefore microprogram minimization is of prime importance.

A half automatic solution is given by TRANSCELL. Special transition function definition instructions may describe transition functions, and minimization directives control the minimization process. Detailed description of the language and its practical application may be found in /L6,L7,L8/.

The TRANSCELL interpreter may produce a minimized microprogram for the hardware and/or may produce a transformed minimized transition function table and a FORTRAN search program

fitting the specialities /caused by minimization/ of the generated table. The search program maps any given combination of a cell's and its neighbours' states onto the continuous address interval of the table containing the values of the next states.

Thus TRANSCELL is a medium level cell microprogramming language, its processor will be built in the simulator in the near future and it may serve as a subsystem in a cellular programming language too.

6. CELLULAR PROGRAMMING

The first *cellular programs* appeared in form of *proofs in constructive mathematics* /simulation of the universal Turing machine and self-reproductive automata/. The traditional way was to embed hardware-like components into the cellular space. This approach to the problem involved construction of *modular, hierarchically structured configurations.*

Here elementary subtasks are realized by less, basically static configurations interconnected for solving subtasks /there is an explicit analogy with subfunctions, subroutines, modules/. Higher levels may be built up hierarchically by interconnections.

For *special purpose applications* as wave-, simulation-spread out, simple self reproduction and picture preprocessing free structured /non-hierarchical, non-modular/ cellular programs may be used that directly do not control the flow of information in the space. Such programs may be characterized as *direct mapping of the problems* /embedding of systems/ into the cellular space. Their application is basically limited for modelling discrete systems /or problems equivalent to them/ where homogeneous local changes dominate the behaviour of the system.

For general purpose computations the embedding and hierarchical interconnections of hardware components give possibility to emulate digital processors. However the emulation of sequentially working hardware /devices like general purpose computers with CPU, memory, etc./ seems to be justified only for theoretic and simulation purposes. Such emulation needs a high number of cells and for general purpose computations it is very uneffective since the emulation factor effects further decreases in the wrong price/performance of the sequential machine. Essentially a cellular space type distributed computing system is very different from a sequential machine consisting of relatively big interacting modules with long data paths. A more effective way would be to emulate systems with more parallelism and short data paths.

The /distributed computing/ nature of the cellular spaces presumes our proposal for effective, modular/parallel cell-programming by means of *embedding into the cellular space processing elements working in parallel on moving data. Different algorithms are to be mapped onto different sets and interconnections of processing elements.*

The principle of modular cellprogramming is preserved. For elementary configurations *software configurations* /processing elements, open subroutines/ are selected.

The cellular space is used in fact as a *distributed computing system* by mapping directly the algorithms into the cellular space in form of static, modular, *hierarchical subconfigurations connected in pipe-line*, e.g. data are moved and transformed along the pipe-line continuously. All the subconfiguration work at the same time ensuring high productivity. The subconfigurations should be of medium or small size, they may be interpreted as open subroutines /e.g. their function may be A=B+C of A=A+L, search from a symbol-table; in general the instruction set of a cellprogramming language/.

In this way cellular processors are suitable to execute a wide class of parallel algorithms /including array processing/. *Direct* /machine code level/ *programming* and microprogramming are enabled by the use of cellular *simulation languages* /ensuring many extra utilities but not generating configurations/. One type of utilities shows possible development to increase the level of programming, namely the library handling routines. It is possible to write subconfigurations by hand, to store them in a library and to call them afterwards.

An assembly type cellprogramming language may consist of an instruction set to call implicitly the members of a set of basic subconfigurations. There exists a difference not only in form but being more than original utility, in addition to the ready subconfigurations at disposal, the call may specify parameters which effect on the called subconfigurations so that the process seems to be a simple configuration generation rather than making a copy from a library.

In a *cell macro assembly language* the user will have the possibility to add his own subconfigurations to the assembler's basic set.

The trend of development points to more sophisticated help automatic and *automatic subconfiguration generations*. In higher level cell-languages the algorithmic description of subfunctions and their relations will be compiled to subconfigurations and their interconnections.

7. SUMMARY

This paper deals with cellprocessors /cellular automata type processors/ being *organic parts in computer architecture* and gives proposals for the structure of very *effective medium speed cellprocessors based on existing technology*.

The task of cellprocessors in the architecture should be to execute procedures effectively computable in parallel at cell level.

The proposed cellprocessor emulates a *finite cellular space* where different groups of cells /represented by cell microprocessors/ may have independently variable transition functions /represented by RAM memories/.

A cell microprocessor consists of an array of /quasi/ cells that may execute the same microinstruction simultaneously. A microprogram /equivalent to a complete transition function, represented by RAM contents/ is executed sequentially. /Therefore microprogram minimization is of prime importance./

A cell may interprete instructions of two types: during the first phase of a transition step the execution of *feature extraction instructions* results in storing the characteristic information about the neighbourhood /in each cell/ and during the second phase the execution of *state assignment instructions* defines the next state /separately in each cell, using the stored neighbourhood description/.

For general computations a *16 state cell* /~100 gates/, for specific purpose applications a *2 state cell* /~10 gates/ have been constructed.

Programs /initial configurations/ and microprograms should be computed and loaded by other processors of the architecture. The information to be processed /input to cellprocessor/ and the results /output from cellprocessor/ are handled in the same way.

General procedures should be programmed by embedding software configurations /open subroutines/ connected in *pipe-line*. Data are moved and transformed parallel along the pipe-line

continuously.

Software support of the above system consists of *simulation languages and a microprogram definition and minimization language*. Higher level cellular languages are under design.

LITERATURE

[1] Neumann, J.: Theory of Automata: Construction, Reproduction, Homogeneity, Part II of "The Theory of Self-Reproducing Automata" ed. A.W.Burks.
University of Illinois, 1966.

[2] Codd, E.F.: Cellular Automata
Academic Press. Inc. New York, London 1968.

[3] Fáy, Gy.: Circuitry Realization of Codd's Cellular Automaton's Cell. Technical Report KGM ISZSZI, 1973.

[4] Takács, D.V.(Mrs): A Bootstrap for Codd's Cellular Space, I.Conference on Computer Science, Székesfehérvár, 1973.

[5] Nourai, Farhad - Sohrab Kasef,R.: IEEE Transactions on Computers Vol.C-24. No.8. August 1975.

[6] Preston, Jr.K.: Use of the Golay Logic Processor in Pattern-Recognition Studies Using Hexagonal Neighborhood Logic
Symposium on Computer and Automata
Polytechnic Institute of Brooklyn, April, 13-15. 1971.

[7] Toffoli, T.: On the Large-Scale Implementation of Cellular Spaces by Means of Integrated-Circuit Arrays.
CNR, Istituto per le Applicazion'del Calcolo, 1972.

[8] Domán, A.: A Model of Parallel Processing.
Információ Elektronika 1975.2. -in Hungarian-

[9] Vollmar, R.: Über einen Interpretierer zur Simulation Zellurarer Automaten.
Angewandte Informatik 6/73.

[10] Brender, R.F.: A Programming System for the Simulation of Cellular Spaces
The University of Michigan, Ann Arbor 1970. /Ph.D.Thesis/

[11] Baker, R. - B.T.Herman: CELIA - A Cellular Linear Iterative Array Simulator.
Proceedings of the Fourth Conference on Applications of Simulation, pp. 64-73 /1970/.

[12] Wu-Hung-Liu: CELIA Users Manual
Dept.of Computer Science, State University of New York at Buffalo, October 1972.

[13] Cellular Spaces, Homogeneous Structures -in Russian - Institute of Mathematical Machines, Warsaw, 1973.

[14] Cellular Automata, Bibliography, KGM ISZSZI Budapest, 1973.

[15] Bibliography on Cellular Automata Papers in Internal Series of KGM ISZSZI 1971-1974.

[16] Vollmar,R.: /manuscript;the bibliography from A.R. Smith III. Introduction and Survey on Polyautomata Theory with supplement/

[17] Rozenberg, G.: Generative Models for Parallel Processes.

[18] Toffoli, T.: Cellular Spaces - An Extensive Bibliography
Dept.of Computer and Communication Sciences. The University of Michigan. 1976.
Publications and internal reports.

L1 Legendi, T.: Simulation and Synthesis of Cellular Automata. Conference "Programming Systems'75".
Szeged, /in Hungarian/.

L2 Legendi, T.: Simulation of Cellular Automata, the Simulation Language CELLAS.
Conference "Simulation in Medical, Technical and Economy Sciences".
Pécs 1975 /in Hungarian/.

L3 Legendi, T.: The CELLAS Cellular Space Simulation Language, /Manuscript in Hungarian 1974./
Martoni,V. - Legendi, T.: CELLAS 0.9 User's Manual, 1974 /in Hungarian/

L4 Czibik, I. - Legendi, T.: User's Manual CELLAS 1.0, 1976. /in Hungarian/

L5 Merényi, E.: The INTERCELLAS Interactive Cellular Simulation Language.
Diploma work. University JATE, Szeged 1975./in Hungarian/.

L6 Zsiros, P.: The TRANSCELL Cellular Automata Transition Function Definition and Minimization Language.
Diploma work, JATE 1975 /in Hungarian/.

L7 Sára, A. - Legendi, T. - Kacsuk, P.: Hardware Design on a 16 state Cellprocessor.
Manuscript in Hungarian, Szeged, 1976.

L8 Legendi, T.: INTERCELLAS an Interactive Cellular Space Simulation Language.
To appear in Acta Cybernetica, Hungarian Academy of Sciences.

L9 Legendi, T. - Hegedüs, Gy. - Pálvölgyi,L.: INTERCELLAS /PDP-8/
User's Manual, 1976. /in Hungarian/.

MECHANICAL ANALYSIS OF HUNGARIAN SENTENCES*

Gy. Hell
Technical University of Budapest,
Institute of Languages
Budapest, Hungary

1. GENERATION AND ANALYSIS OF SENTENCES

1.1 Inapplicability of finite state grammars

In the generation process of Hungarian word forms a finite state grammar has been used. On its basis all the Hungarian word forms could be produced and analysed. The finite state diagram gives good representation of how a word form is produced and makes it considerably easy to write a program according to it.

The finite state diagram produces elements in successive order /from left to right/ so that the production of an element is the result of the preceding item and is dependent on it. This means that such a generation process is very helpful when we have to describe the relation between two successive elements but it cannot be used /or only in a very complicated way/ if we want to express the relation between not succesively ordered elements. E.g.:

---

* The first part of this paper /Mechanical Analysis of Hungarian Word Forms/ has been published in issue X. of CL & CL, pp. 125-134.

1/ Végre *megvan* a megoldás.
(At last we have the solution.)

2/ Végre *meg van* oldva a feladat.
(At last the problem has been solved.)

3/ *Nemcsak* megoldotta a feladatot, *hanem* meg *is* indokolta a megoldást.
(He not only solved the problem but he explained the solution too.)

In sentence No.1 the morphemes *meg* and *van* not only follow each other but they are also connected into one word unit. In sentence No.2 the morpheme *meg* forms a word not with the immediately following *van* but with *oldva* (this fact is expressed by orthography too). In sentence No.3 the conjunction *hanem is* is in close relation with the first word of the sentence *nemcsak* and depends on it entirely. As such constructions very often occur in (Hungarian) sentences, a phrase structure grammar (PS) will be used instead of a finite state grammar for describing the structure of the sentences.

1.2 Phrase structure grammar for Hungarian syntactical constructions

PS rules describe sentences by giving the immediate constituents for different levels of analysis. A sentence may have the following constituents:

4/ A művész uj képet festett.
(The artist a new picture painted.)

levels	constituents					
I.	A művész	+	új képet festett.			
	NP	+	VP			
II.	A + művész		új képet	+	festett	
	D + N		NP	+	V	
III.			uj	+	képet	
			Adj	+	N	

The PS rules expressing the structure of the sentence:

S ⟶ NP + VP
VP ⟶ NP + V
NP ⟶ /D/+/Adj/+N
V ⟶ fest
N ⟶ művész
kép
Adj ⟶ új
D a

(Symbols in parentheses denote facultative use.)

In Hungarian sentences we have three different main constituent structures [1] :

a/ verbal structures (VER)
b/ nominal sturctures (NOM)
c/ adverbial structures (AD)

They can be described by the following PS rules:

I. VER $\left[\begin{array}{l} \text{Aux Inf} \\ \left(\text{Aux}\right) \begin{array}{l} \text{V} \\ \text{V}_n \\ \text{V}_a \end{array} \\ \text{S}_v \end{array}\right]$

II/a. NOM ⟶ $\left[\begin{array}{l} \text{PN} \\ \text{DNO} \\ \text{S}_N \end{array}\right]$

b. PN ⟶ $\left[\begin{array}{l} \text{Pn}_a \\ \text{Pn}_b \\ \vdots \\ \text{Pn}_m \end{array}\right]$

c. DNO ⟶ (DET) NON

d. DET ⟶ $\left[\left[\begin{array}{l} \text{D} \\ \text{Pn}_d \end{array}\right] \quad \text{Nu}\right]$

e. NON ⟶ $\left[\begin{array}{lll} & & \text{NO} \\ & (\text{NO}^B) & \text{NO}^A \\ \text{DNO}^{BS} & (\text{DET}) & \text{NO}^{BK} \end{array}\right]$

$$\text{f. NO} \longrightarrow (\text{ADJ})\ (\text{PA})\begin{bmatrix}\text{ADJn}\\ \text{ADJ}\end{bmatrix}\ \text{N}$$

$$\text{g. ADJ} \longrightarrow (\text{NON})\ (\text{AD})\ \text{Adj}$$

$$\text{h. ADJn} \longrightarrow (\text{ADJ})\ \text{An}$$

$$\text{i. PA} \longrightarrow (\text{NOM}_1, \text{NOM}_2, \ldots, \text{NOM}_n)\ (\text{AD})\ \text{P}$$

III/a.

$$\text{ADV} \longrightarrow \begin{bmatrix}\text{GER}\\ \text{AD}\\ \text{NOM}_A\\ \text{S}_A\end{bmatrix}$$

$$\text{b. AD} \longrightarrow (\text{ADv}^B)\ \ \text{ADv}^A$$

(Two or more symbols placed under each other in parentheses denote the obligatory use of one of them.)

The PS rules contain two different kinds of symbols. One of them appears only on the right side of the rules while the other can be seen on both sides.

On the right side we have the following symbols: D, Pn_a, $\text{Pn}_b, \ldots, \text{Pn}_m$, Pn_d, Nu, N, Adj, An, Adv, P, Aux, Inf, V, V_a, V_n. These symbols stand for only one word in the sentence and represent word categories. The other symbols signify categories composed mostly of two or more words as larger constituents of the sentence. Their relations in the rules express the relations which the represented word groups have in the sentence.

1.3 Dependency grammar of the sentence structure

If we apply the PS rules to sentence No.4 we obtain the categories characterising smaller and larger units, e.g.:

4/a. ///A/$_{\text{D}}$/művész/$_{\text{N}}$/$_{\text{DNO}}$/$_{\text{NOM}}$////új/$_{\text{Adj}}$/képet/$_{\text{N}}$/$_{\text{NO}}$/$_{\text{NOM}}$
/festett/$_{\text{V}}$/$_{\text{VP}}$/$_{\text{S}}$

On the one hand this form of description gives a good picture of the hierarchical structure in the sentence but on the other hand it places the units of a larger category on the same level as if they had the same role in constructing a constituent of higher order. Our intuitive knowledge about the sentence and also a more detailed grammatical description says that e.g. the units *új* and *képet* (new and picture resp.) have different roles in building the NO phrase. From among the two words the first one (*új*) is obviously less important (in constituency) than the N, since it is possible to have an NOM construction without it, e.g.: $///A/_{D}/művész/_{N}/_{NOM}$. This fact can relatively simply be expressed by the help of a dependency description. As dependency grammars give dependency relations between the words of a sentence, we can start with the phrase structure grammar and take the rules which contain word category symbols. Our dependency rules will differ from the PS rules only in that they make explicit which of the two (or more) constituents is the main element and where it is placed in the construction.

Taking the PS rules with word category symbols we make the following changes ② :

$$
\begin{aligned}
&NO \rightarrow N \;\; //ADJ/,/PA/,/ADJ/,+\_/\\
&ADJ \rightarrow Adj//NOM/,/AD/,+\_/\\
&ADJn \rightarrow An \;//ADJ/,+\_/\\
&PA \rightarrow P \;\; //NOM_1, NOM_2,\ldots,NOM_n/,/AD/,+\_/\\
&AD \rightarrow ADv//Adv^M/,+\_/
\end{aligned}
$$

Rule II/b in the PS rules has only one element and rule II/d has two elements with equal status. The corresponding dependency rules are:

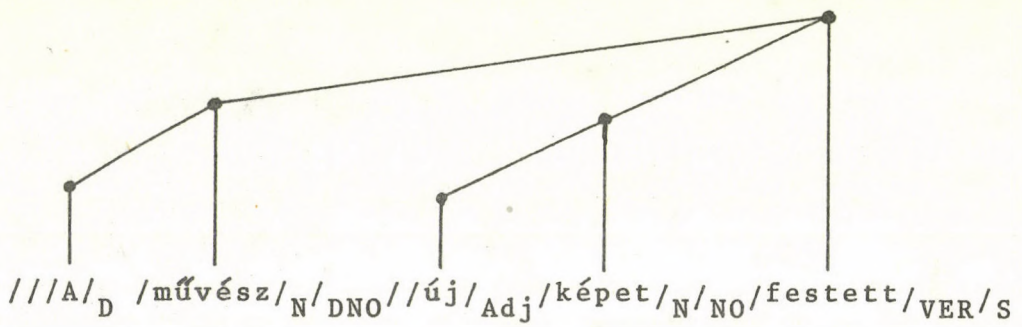
$$DNO \rightarrow NON \;/ \begin{pmatrix} D \\ Pn_d \end{pmatrix} ,/Nu/,+\_/$$

i.e. the categories D, Pn_d, Nu have one of the possible NON categories as main structure element.

All these rules express dependency relations within different NOM, ADJ and AD structures. Since a sentence is a unit consisting of different such structures we have to add a new rule to the previous ones expressing the subordination of all obtained structures to the verb (predicate).

$$S \rightarrow VER \left[\begin{matrix} NOM_1, NOM_2, \ldots, NOM_n \\ ADV_1, ADV_2, \ldots, ADV_m \end{matrix} \right]$$

Now we can give the representation of sentence no.4 in a dependency tree according to the rules of dependency grammar:



In this representation the words of the sentence correspond to nodes in the tree, and the verb is on the root. The fact that a word belongs to a special kind of construction is expressed differently in different languages. In most of the inflected languages the agreement in case, gender and number, and special "governing rules" refer to the connection between the elements in the same construction. In Hungarian the situation is somewhat different. Hungarian has no grammatical category of gender and there is no agreement of case and number between adjectives and nouns. The most important feature of Hungarian NOM, ADJ and ADJn constructions is expressed by the fact that the main element follows the dependent members. Beside semantical restricitions there is a strict ordering rule for word categories in the constructions and a NOM construction usually begins with the determiner of the main element. This fact is expressed in our PS and dependency rules; thus when the mechanical analysis of a word sequence is to be performed, we

have to decide whether the given word as main element forms a construction with the/a previous one or it is dependent on the next item, or it is independent of them.

2. THE PROCESS OF ANALYSIS

In the morphological analysis word forms are reduced to stems and endings. On this level there is no necessity to define what categories the word stem and the ending belong to. From the point of view of analysis e.g. it is not necessary to know that in the homographic form *várnak* (they await me - to the castle) the stem *vár* (to wait - castle) is simultaneously a noun and a verb and the suffix *-nak* can be a verb ending and a noun ending too. The recognition of stem and ending is nevertheless correct. If morphological analysis is taken as first step to syntactical analysis, then we have to do more. As the recognition of syntactical units requires information about word categories, the vocabulary for morphological analysis must contain information of this kind too.

2.1 Organization of the mechanical dictionary

The organization of the machine dictionary is closely connected with the process of analysis. It is desirable to arrange the dictionary entries in a way which reduces the time required for looking up. This was aimed at by a double organisation of data.

1. Dictionary entries /word stems, endings, formatives/ are arranged according to their length,
2. within a given length, entries are located in alphabetic order.

The dictionary of word stems contains the following items of information:

1. category of word class,
2. subcategory of word class,
3. information about selectional restrictions.

Beside traditional word classes as:VERB/1/, NOUN/2/, ADJECTIVE /3/, NUMERAL/4/, PRONOUN/5/, ARTICLE/6/, POSTPOSITION/7/, ADVERB/10/, VERBAL PARTICLE/22/ PARTICIPLE/23/, CONJUNCTION/11/ the dictionary contains the class

An = a noun stem with the adjectival formative -ú/-ű, or
a noun and its postposition with the adjectival formative -i.

The homographic categories of the dictionary are as follows ③:

VERB/NOUN /13/
NOUN/ADJECTIVE /14/
NOUN/PRESENT PARTICIPLE /15/
ADJECTIVE/PRESENT PARTICIPLE /16/
VERB/PAST PARTICIPLE /20/
VERB/ADVERB /21/
VERBAL PARTICIPLE/ADVERB /24/
VERB/NOUN/PAST PARTICIPLE /25/

The dictionary of endings has the following information stored:

1. word class of the respective stem,
2. grammatical features of the ending.

Verb endings may contain grammatical information as follows:

1. person of the finite form,
2. number of the finite form,
3. tenses,
4. definiteness or indefiniteness,
5. modality,
6. person of the definite object,
7. number of the definite object.

Noun endings give information about:

1. case ending /27 possible cases/,
2. number,
3. number and person of possessor,
4. indication about being a possessor,
5. indication about having a postposition.

The dictionary look-up procedure consists in identifying the word stem and not the ending. This strategy is necessary in Hungarian because most of the words in a sentence have letters and letter combinations at their end which can be mistaken for endings. /Out of the 38 letters in the Hungarian alphabet only 12 do not correspond to any ending./ In the searching procedure the longest indentified part is taken for a stem, the remaining part is supposed to be and ending in the first round but if it does not correspond to an item in the list of endings, the program starts again looking for a possible stem. /See block diagram, Fig.1 /

As stems and/or endings may be homographic, it is necessary to correct the information given to them from the dictionary. This correction is carried out according to a matrix where the possible combinations of stems and endings are listed. /See Table I/

2.2 Identification of syntactical units

The syntactical analysis of a simple sentence begins with the identification of the predicate. It is not too difficult to find the predicate when it is a finite verb form and there is no other word in the sentence which, by its form, could be mistaken for a verb. /In Hungarian the past participle has the same form as the 3rd person singular of the past tense./

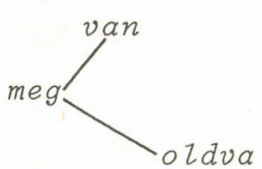
In principle a syntactical analysis does not necessarily begin with looking for the predicate, but some practical considerations make it reasonable to use such a strategy: by identification of the predicate a division of the sentence is given what provides valuable information for the later phases

of the analysis.

If the sentence under analysis has no coordinate constructions, then all its elements are connected by dependency relations. Difficulties may arise, nevertheless, in cases of separated units, discontinuously arranged or analytic forms. What sufficient motivation do we have to say that the verb stem is dominated by its prefix, or, the prefix is dependent on the verb? Instead, it is more reasonable to consider such disconnected elements as units represented by a single node in the dependency tree, but extended in the sentence into two or more words. In this way we have all the necessary information in one node what allows us to establish the relations upward and downward of the tree ④. /The correct morphological analysis requires anyway the unification of the two items./

Nevertheless, extensions in themselves can also be considered as elements with dependency relations between them. So, e.g. the analytic verb form of state or condition in sentence No.2 gives the following dependency structure:

Végre *meg van oldva* a feladat.



The machine representation of the dependency relation of a sentence corresponds to a matrix where the rows contain the dependent elements and the columns the superordinate nodes ⑤. For sentence No.4 the dependency matrix has the following form:

A művész új képet festett.
1 2 3 4 5

	5	2	4
2	1	0	0
1	0	1	0
4	1	0	0
3	0	0	1

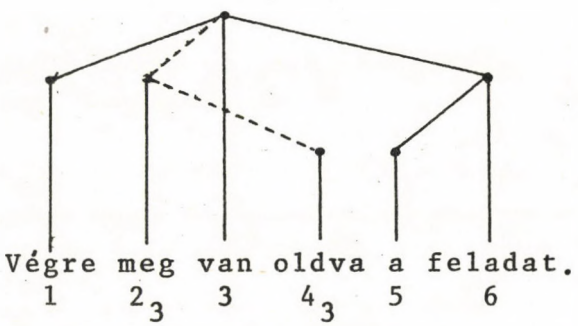
The dependency matrix and extension matrix resp. for sentence No.2 have the forms:

	3^{+}	6
1	1	0
6	1	0
5	0	1

③	3	2
2	1	0
4	0	1

In the matrices a special marking is given to the element representing the comprehending node.

The corresponding dependency tree:



Words reduced to one node are indexed by the main word in the extension.

Beside disconnected and analytic word forms also other constructions are regarded as extensions: nouns and their postpositions /this way they have the status equal to that of the suffixed nouns/, negative and modifying particles with the modified element, finite verb forms with infinitive.

In an extension the members do not necessarily have a strict dependency relation. So coordinate constructions, too, are handled as extensions with the conjunction as main element.

The matrix for the sentence No.5:

Az utcában régi és új házak állnak.
1 2 3_4 4 5_4 6 7
(In the street new and old houses stay.)

have the following form:

	7	6	2
6	1	0	0
2	1	0	0
4+	0	1	0
1	0	0	1

Dependency matrix

④	4
3	1
5	1

Extension matrix

From among the syntagmatic /=constituent/ structures the noun construction is evidently the most complicated. It has the most numerous possibilities of realization and involves great difficulties for the analysis. The difficulties arise from the fact, that in a noun construction the number of different participating categories is fairly large. So, beside Adj, An, D, Nu, P and Pn still other (major) construction categories can be found in the PS rules II/a - II/i:

$NO/NON \rightarrow N^{BS}/NO^{BS} + N^{BK}/NO^{BK}$	in II/e
$NO/NON \rightarrow N^{B}/NO^{B} + N^{A}/NO^{A}$	in II/e
DNO	in II/c
PA	in II/f
AD	in II/g

Considering that in a NOM construction some N or NO belong to an Adj or P which are dependent members of an N, we must add to the above list still such categories as:

$$NO_A/NON_A$$
$$NO_P/NON_P$$
$$NO_A^{BK}$$
$$NO_P^{BK}$$

where the subscripts indicate dependency relations.

Now, it is possible to construct matrices for the possible arrangement of a nominal construction too, in a similar way as it has been done for the correction of stem and ending information. The only difference will be that in this case we have to take the different work and construction categories instead of the stem and ending categories and to check which of them has been realized. A section of this possibility matrix is shown in Table II.

In the sentence No.6:

> A kielégitő gépi szövegfeldolgozás nem kivánja a formális és tartalmi elemzés közötti különbségek megszüntetését.
>
> /The satisfactory mechanical text processing does not require the abolition of the differences between the formal and the content analysis./

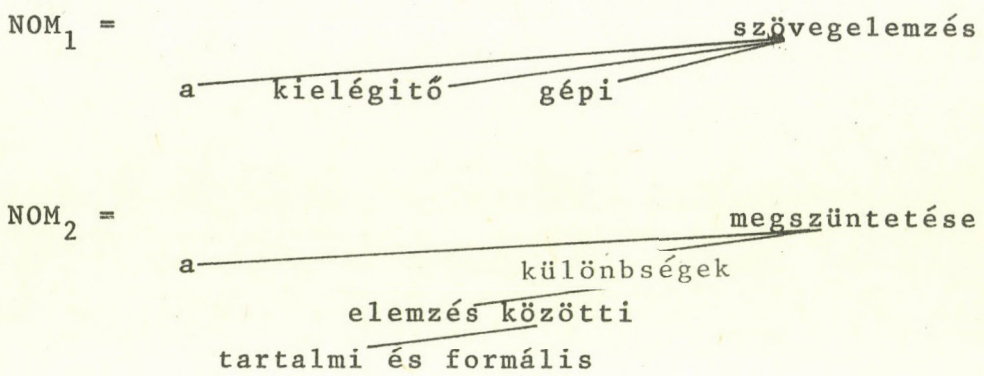
the analysis of the enlarged object construction /a formális és tartalmi elemzés közötti különbségek megszüntetését/ is carried out in following steps:

analyzed item /=i/	categ. of i	input state			output state		
		NO	NO^{BK}	An	NO	NO^{BK}	An
megszüntetését	N^{BK}	0	0	0	0	1	0
különbségek	N^{BS}	0	1	0	1	0	0
elemzés közötti	An	1	0	0	1	0	1
formális és tartalmi	Adj	1	0	1	1	0	1
a	D	1	0	1	1	0	0

As a result of this analysis we have got the major constituents of the sentence /its syntagmatic units/ and the dependency relations within them:

VER= kívánja — nem

is an extended construction



2.3 Syntactical analysis

At the end of the syntagmatic /constituent/ analysis we have one or more noun constructions and/or adverbial constructions beside a verbal construction. Major noun syntagmas are described /in the PS rules/ in a way that the head of the construction always directly depends on the verb in the sentence ⑥. Now, in the syntactical analysis we have to define what different relations the nominal and adverbial constructions have to the verb. According to traditional grammars we distinguish the following syntactic categories:

a/ subject, /S/
b/ direct object, /O/
c/ dative object, /D/
d/ indirect object, /IO/
e/ instrumental. /I/
f/ adverbial of direction, /+el,-el,+on/
g/ adverbial of state,
h/ time adverbial,
i/ place adverbial,
j/ adverbial of manner.

The Hungarian language is fairly explicit in expressing various syntactical relations by different endings: subject and direct object, dative and indirect object, different adverbials are usually clearly distinguished by different endings.

Nevertheless in spite of the numerous case endings in Hungarian /27 endings/, suffixes are homonymous and the same ending may express different syntactical functions. E.g.:

Sentence No.6:

> Hibás alkatrészt javított a munkahelyén.
> /A defected machine part he repaired in his working place./

Sentence No.7:

> Tíz másodpercet javított az eredményén.
> /By 10 seconds he improved his result./

In both sentences Nos. 6 and 7 we have two complements to the same verb (*javított*): one complement with the ending *-t* and another complement with the ending *-án/-én*. It is clear from the content of the sentence that in No.7 the complement *másodpercet* is not a direct object as *alkatrészt* in No.6 but a time adverbial, while *eredményén* is not a real location as *munkahelyén* in No.6.

It can be stated that the different syntactical meanings of the complements derive from the fact that the two nouns express two different semantic categories: *alkatrész* is a real object, *másodpercet* is a time category, *munkahely* is a place category and *eredmény* is an abstract noun not capable of expressing a real place. As lexical meaning is really important in the decision process of what syntactical function a word may have in the sentence, stems must possess information about what semantical categories and subcategories they belong to.

The difference in the syntactical meanings of the words with the same ending becomes clear by the fact that sentence No.7 can be transformed into

Sentence No.7/a

Tíz másodperccel javította az eredményét.

In this sentence the case endings of the noun phrases have changed: *másodperccel* has an instrumental ending and *eredményét* has the ending *-t* and is a direct object. The meaning of the sentence has not changed. No similar transformation of sentence No.6 is possible.

Now, if we don't ask whether the two sentences express exactly the same meaning it is true that *-t* vs. *-vel* and *-án/-én* vs. *-t* are free variants and so they must belong to the same category. Moreover, the correctness of such a classification can be proved by the following: sentence No.7 allows the transformation into sentence No.7/b:

Sentence No.7/b

Tíz másodperccel javított az eredményén.

After that we have the combinations:

"time"	"object"
-t	*-án/-én*
-vel	*-t/-én*

Enlarging the sentence with a new complement we can say:

Sentence No.8/a

Tíz másodpercet javított az eredményén az új pályán.
/By 10 seconds he improved his result on the new track./

Sentence No.8/b

Tíz másodperccel javította az eredményét az új pályán.

Sentence No.8/c

Tíz másodperccel javított az eredményén az új pályán.

While "time" and "object" relations may vary freely in the sentence, the ending *-án/-én* on the word *pálya* remains unchanged: it expresses a category definitely different from *-t/-én*.

In the syntactic analysis we distinguish two types of case endings:

a/ simple case endings,
b/ allographic case endings as free variants.

For the characterization of syntactic relations we use not only the semantic categorization of verbs and their object nouns, but the property of the verb to have complements with simple and allographic case endings as well. We suppose that allographic endings express the same deep structure meaning which finds its realization in different surface case forms ⑦.

The property of a verb to have complements with different case endings is stored in the machine dictionary as its selectional restriction. /In Table III the selectional restrictions of a few words are given./ On the basis of this information the noun constructions around the verbal predicate can be identified as complements with different syntactical meanings. It may happen that selectional restrictions to the predicate are not complete and they have to be corrected or completed in the analysis. This is the case in sentences with an accusative and infinitive construction.

Sentence No.9

Hallottam a barátomat énekelni az operában.
/I heard my friend to sing in the opera./

The reduction of the extension *hallottam énekelni* is carried out in the presyntactic phase of the analysis. At this point

a computation of the selectional restrictions to the two verbs is also necessary. The selectional restrictions to the two verbs are:

	P	D	S/O	S/D	...	O	...	O/-el	D	I	...	Inf
hallani (hear)	+				...	+	...	+		+	...	+
énekelni (sing)	+				...	+	...	+	+		...	+

After the computation process the predicate of the sentence has in its selectional restrictions an actor/object:

	P	D	S/O	S/D	...	O	...	O/-el	D	I	...	Inf
hallottam énekelni	+		+		...	+	...	+	+	+	...	

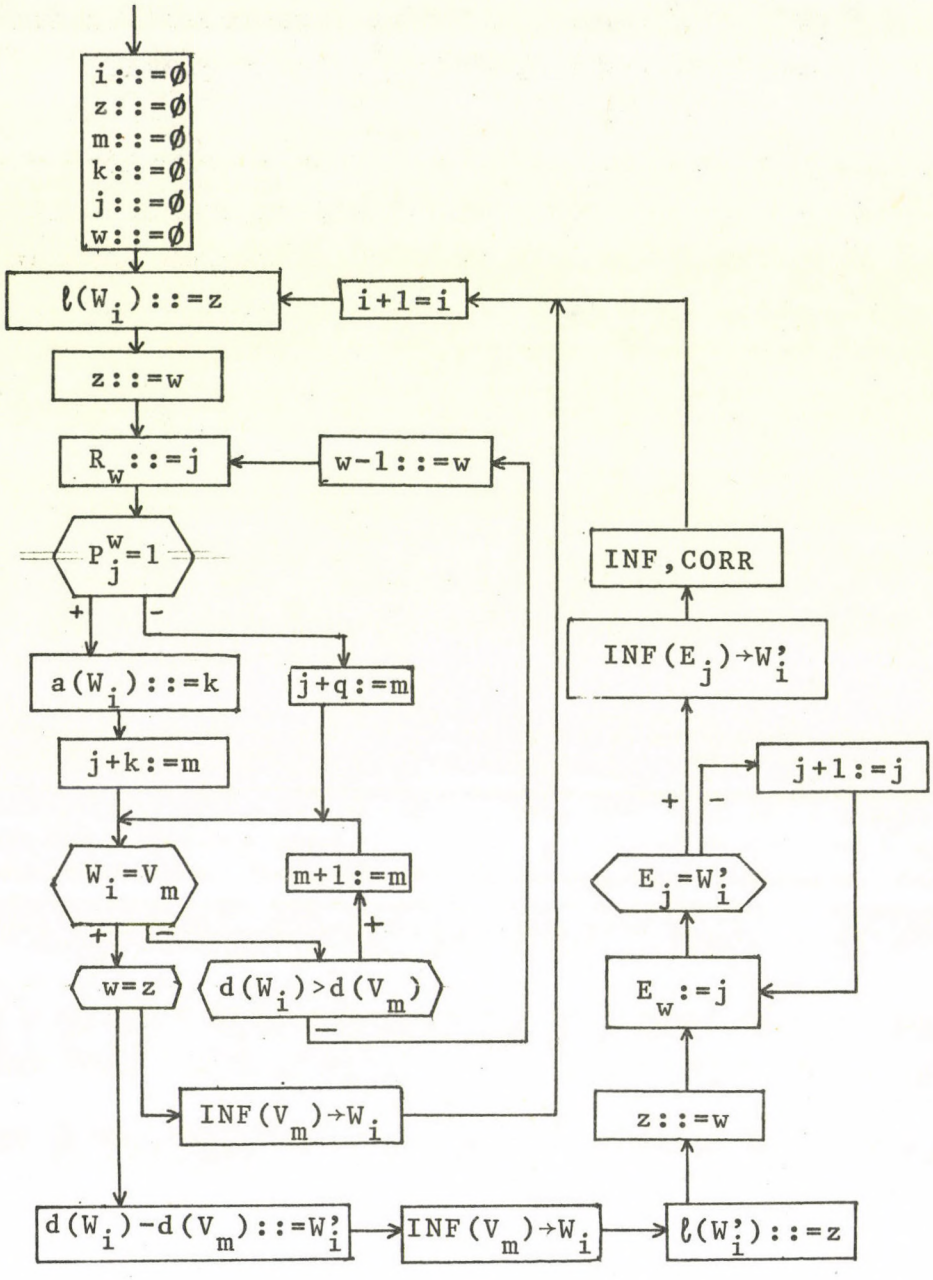
3. STRATEGY OF SENTENCE ANALYSIS

The mechanical syntactical analysis described in the previous pages works only on simple sentences with no coordinated structures and ellypses. An extension of the system, to make it capable to analyse complex sentences implies serious problems /even if coordination and ellypsis are excluded/. It seems reasonable that in a complex sentence the analysis of constituent structures should not immediately follow after morphological procedures. As a first step it is obviously necessary to decide what structure the whole sentence possesses. Is it a compound sentence with coordinated members or is it a complex with clauses subordinated according to different functions and order? In deciding it, it is necessary to analyse the whole sentence for its coordinated members and clauses.

Arrangement of sentence connectives, that of punctuation marks and predicate words can give valuable information in this process. /In Hungarian orthography clauses and constituent sentences have to be separated by commatas./ Only if this

procedure gives reliable clues about the arrangement of clauses and constituent sentences, the analysis of syntagmatic constituents can be initiated.

The analysis was carried out on a RAZDAN-3 computer of the Computing Center of the Universities. The program was written in ASTRA 2, a compiler language with special macros for natural text processing. The output form is given in Fig.2.



V = vocabulary of stems
E = vocabulary of endings
W_i = word processed
$\ell(W_i)$ = number of characters in W_i
R = register of V-blocks according to $\ell(W_i)$
$a(W_i)$ = coded numerical value of the first letter in W_i
P^w = alphabetic register of word stems with the length w
$d(W_i)$ = numerical value of the coded word form
q = constant

Fig.1 Block diagram of morphological analysis

		1	2	3	4	5
endings	1	+				
	2		+			
	13			+	+	+
stems	1	+		+		
	2		+		+	
	3		+		+	
	13	+	+			+
	14		+		+	
	15		+		+	
	17	+	+		+	
	20	+			+	
	21	+		+		
corrested cat.		1	2	1	2	13

Table I. Possibility matrix of homographic stems and endings

NON categ.	P_n	+	+	+																
	N	+	+	+																
	N^A	+	+	+																
	N^B				+	+														
	N^{BK}	+		+																
	N^{BS}						+	+	+	+										
	A_n										+	+	+	+	+	+				
	$A_n{}^{BK}$										+	+	+	+	+	+				
	A_j																+	+	+	+	+	
	P																+	+	+	+	+	
	Ad^A																					...
	Ad^B																					...
	Nu																					...
	D																					...
input state	NO		+	+		+		+	+	+	+		+	+	+	+	+		+	+	+	...
	NO^A		+		+																	...
	NO^{BK}						+					+						+				
	NO_P			+						+			+						+			
	NO_P^A					+																
	NO_P^{BK}							+						+						+		
	NO_A		+							+					+						+	
	NO_A^{BK}								+							+						
	An^{BK}									+												
	An^A		+			+																
	An		+																			
	Aj			+					+	+					+	+						
	P			+				+		+			+	+								...
operations		1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...

Table II. Possibility matrix of NON constructions

OPERATIONS IN THE NON POSSIBILITY MATRIX

1. delete input state; output: $NO/NO^{A}/NO^{BK}$
2. d/i/ = NO^{A}; output: NO
3. d/i/ = NO_{P}^{A}/An^{A}; output: $NO+NO_{P}/An$
4. d/i/ = NO^{BK}; output: NO
5. d/i/ = NO_{P}^{BK}; output: $NO+NO_{P}$
6. d/i/ = NO_{A}^{BK}; output: $NO+NO_{A}$
7. d/i/ = An^{BK}; output: $NO+NO_{P}/NO_{A}+An$
8. d/i/ = NO; output: $NO+An/An^{BK}$
9. d/i/ = NO^{BK}; output: $NO+An^{A}/An^{BK}$
10. d/i/ = NO_{P}; output: $NO+NO_{P}+An/An^{BK}$
11. d/i/ = NO_{P}^{BK}; output: $NO+NO_{P}^{BK}+An/An^{BK}$
12. d/i/ = NO_{A}; output: $NO+NO_{A}+An/An^{BK}$
13. d/i/ = NO_{A}^{BK}; output: $NO+NO_{A}^{BK}+An/An^{BK}$
14. d/i/ = NO; output: NO+Aj/P
15. d/i/ = NO^{BK}; output: $NO^{BK}+Aj/P$
16. d/i/ = NO_{P}; output: $NO+NO_{P}+Aj/P$
17. d/i/ = NO_{P}^{BK}; output: $NO+NO_{P}^{BK}+Aj/P$
18. d/i/ = NO_{A}; output: $NO+NO_{A}+Aj/P$

.
. d/i/ = main element of i
.

	P	D	S/O	S/D	S/I	S/+el	O	O/D	O/I	O/+el	O/-el	O/on	D	D/I	I	+el	-el	on	Inf
leáll (to stop)		+			+										+		+		+
közlekedik (run, communicate)	+	+			+										+				
beállit (set in)	+						+						+			+			
gyülekezik (gather)	+																		
kitelepít (displace)	+						+									+			
megy (go)	+	+														+	+	+	+
kérdez (ask)	+						+				+						+		
elmond (tell)	+						+						+				+		
olt (put out)																			
⋮																			

Table III. Selectional restrictions to verbs

```
KOROSLADAYNY EYS SZEGHALOM KOZOTT LEAYLL A VASUUTI FORGALOM .

KOROSLADAYNY/   EYS/   SZEGHALOM/   KOZOTT/   LEAYLL/   A/   VASUUTI/   FORGALOM/   .

AYLLIYTAAYNY:     LEAYLL
         0000000004430UU1
         0000320004420UU0


A KEYT KOZSEYG KOZUIT KOZLEKEDOY EGYETLEN SZERELVEYNYT BEAYLLIYTJAYK A LADAYNYI PAYLYAUDVARRA
.

A/   KEYT/   KOZSEYG/   KOZOTT/   KOZLEKEDOY/   EGYETLEN/   SZERELVEYNY/T   BEAYLLIYT/
JAYK   A/   LADAYNYI/   PAYLYAUDVAR/[illegible]   .

AYLLIYTAAYNY:     BEAYLLIYTJAYK
         0143000021930UU1
         0000504044740UU0


   GYEREKEK EYS ASSZONYOK GYYLEKEZNEK .

A/   GYEREK/EK   EYS/   ASSZONY/OK   GYYLEKEZ/NEK   ..

AYLLIYTAAYNY:     GYYLEKEZNEK
         0000300004930UU1
         0000500000U00UU0


A NAGY MAYRTON UTCA LAKOUIT KITELEPIYTIK .

A/   NAGY/   MAYRTON/   UTCA/   LAKOU/IT   KITELEPIYT/IK   .

AYLLIYTAAYNY:     KITELEPIYTIK
         0143000016930UU1
         00004040UU00UUU0


KOROSLADAYNY EYS SZEGHALOM KOZOTT LEAYLLT A VASUUTI FORGALOM .

KOROSLADAYNY/   EYS/   SZEGHALOM/   KOZOTT/   LEAYLL/T   A/   VASUUTI/   FORGALOM/
.

AYLL YTAAYNY:     LEAYLLT
         000000104440300U1
         0000320004420UU0
```

Fig. 2

```
KQRQSLADAYNY EYS SZEGHALOM KQZQTT LEAYLL A VASUYTI FORGALOM .
KQRQSLADAYNY/ EYS/ SZEGHALOM/ KQZQTT LEAYLL/ A/ VASUYTI/ FORGALOM/ .
AYLLIYTMAYNY:    LEAYLL
        0000000004230001
        0000320004220000

A KEYT KQZSEYG KQZQTT KQZLEKEDQY EGYETLEN SZERELVEYNYT BEAYLLIYTJAYK A LADAYNYI PAYLYAUDVARRA
A/ KEYT/ KQZSEYG/ KQZQTT/ KQZLEKEDQY/ EGYETLEN/ SZERELVEYNY/T BEAYLLIYT/
JAYK A/ LADAYNYI/ PAYLYAUDVAR/RA .
AYLLIYTMAYNY:    BEAYLLIYTJAYK
        0143000021330001
        0000504044440000

   GYEREKEK EYS ASSZONYOK GYWLEKEZNEK .
A/ GYEREK/EK EYS/ ASSZONY/OK GYWLEKEZ/NEK .
AYLLIYTMAYNY:    GYWLEKEZNEK
        0000300004330001
        0000500000000000

A NAGY MAYRTON UTCA LAKOYIT KITELEPIYTIK .
A/ NAGY/ MAYRTON/ UTCA/ LAKOY/IT KITELEPIYT/IK .
AYLLIYTMAYNY:    KITELEPIYTIK
        0143000016330001
        0000404000000000

KQRQSLADAYNY EYS SZEGHALOM KQZQTT LEAYLLT A VASUYTI FORGALOM .
KQRQSLADAYNY/ EYS/ SZEGHALOM/ KQZQTT/ LEAYLL/T A/ VASUYTI/ FORGALOM/
AYLLIYTMAYNY:    LEAYLLT
        0000001044030001
        0000320004220000
```

Copy of original Fig.2; reproduced by type-writer

NOTES:

① The concept of main constituents given in this paper differs from the "standard" description /Chomsky, N. 1965/ where NP and VP are the two main constituents. The reason for this lies in the fact that in Hungarian sentences subject and predicate parts are usually mixed up and a fixed order of NP and VP is not characteristic. A similar position is taken by Deme, L. /1971/.

② The notation is taken from Hays, D. /1964/.

③ Number in parentheses are code numbers used in the program.

④ The reduction of extended parts turns non-projective branches into projective ones.

⑤ See Zierer, E. /1970/.

⑥ Noun constructions complementing not the verbal predicate but the preceding noun, are not characteristic for the Hungarian language.

⑦ This system of selectional restrictions remembers the valence theory /see: Helbig, G. 1969/ and Fillmore's case grammar. The case categories here are surface structure cases and the "deep structure cases" are closely connected with surface forms. "Deep cases" appear only by verbs which can be used with the same noun in two different surface case forms, expressing the same meaning.

REFERENCES

[1] Chomsky, N: Aspects of the Theory of Syntax; The MIT Press, Cambridge, Massachusetts, 1965.

[2] Deme, L.: Mondatszerkezeti sajátosságok gyakorisági vizsgálata (Magyar szövegek alapján); Akadémiai Kiadó, Budapest, 1971.

[3] Fillmore, J.: The Case for Case, in: Bach and Harms (eds.), Universals in Linguistic Theory, New York 1968.

[4] Hays, G.: Dependency Theory: A Formalism and Some Observations; Language XL, 511-525, 1964.

[5] Helbig, G.: Einführung in die Valenztheorie, in: G.Helbig and W.Schenkel (eds.), Wörterbuch zur Valenz und Distribution deutscher Verben; Leipzig, 1969.

[6] Jávorszky, Gergely: ASTRA - A RAZDAN-3 gépikódu programozásának programnyelve, in: Tájékoztató 9. (1972), Egyetemi Számitóközpont, Budapest.

[7] Zierer, E.: The Theory of Graphs in Linguistics, Mouton the Hague, 1970.