XI

CL & CL

Computational Linguistics and Computer Languages

computer
and
automation
institute
hungarian
academy
of sciences

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

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.

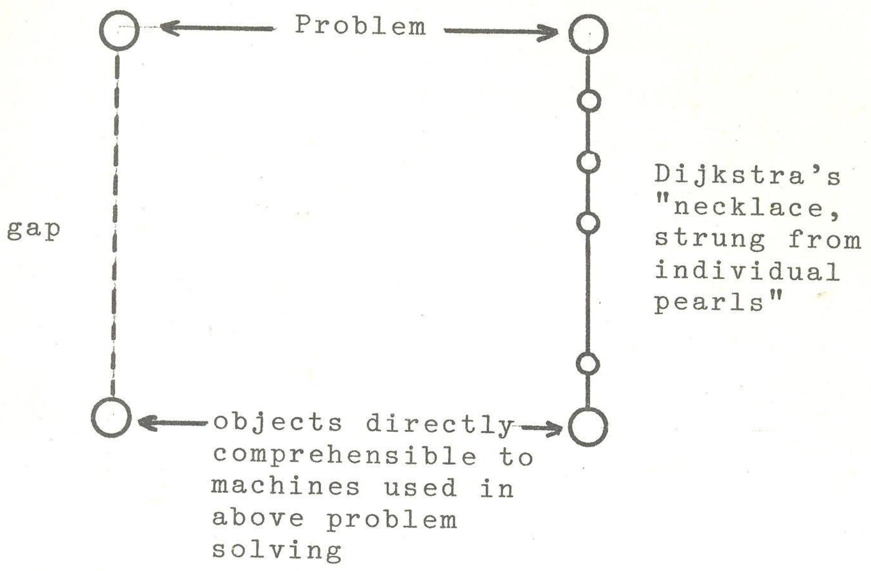

Problem
gap
Dijkstra's "necklace, strung from individual pearls"
objects directly comprehensible to machines used in above problem solving

Fig.1a *The "complexity problem*

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

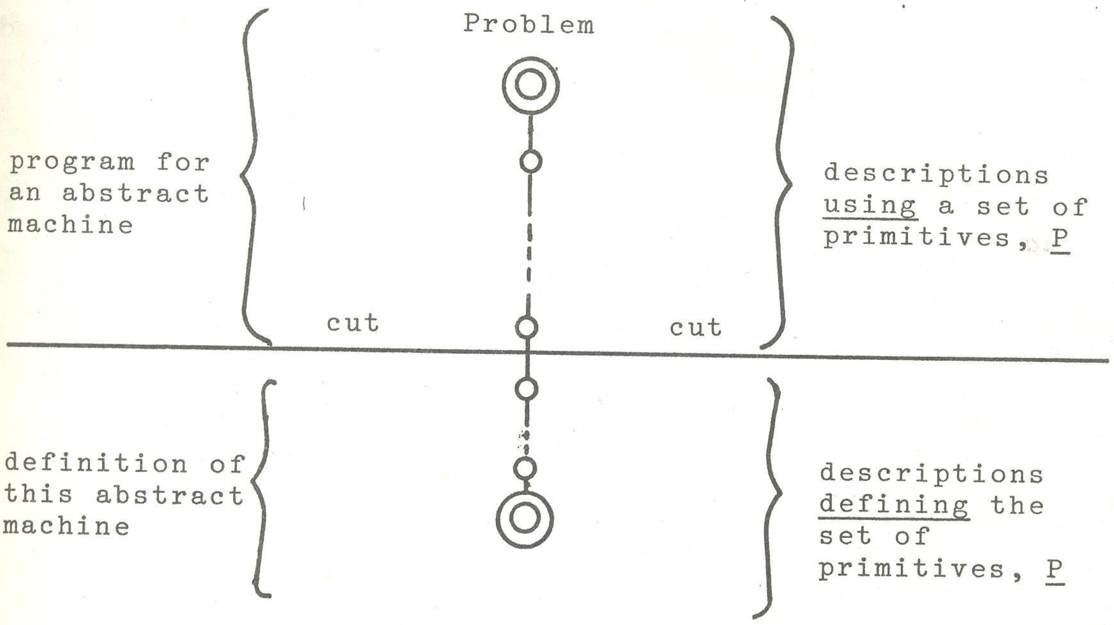

Problem
program for an abstract machine
descriptions using a set of primitives, P
cut
cut
definition of this abstract machine
descriptions defining the set of primitives, P

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 (lc *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.
O	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.

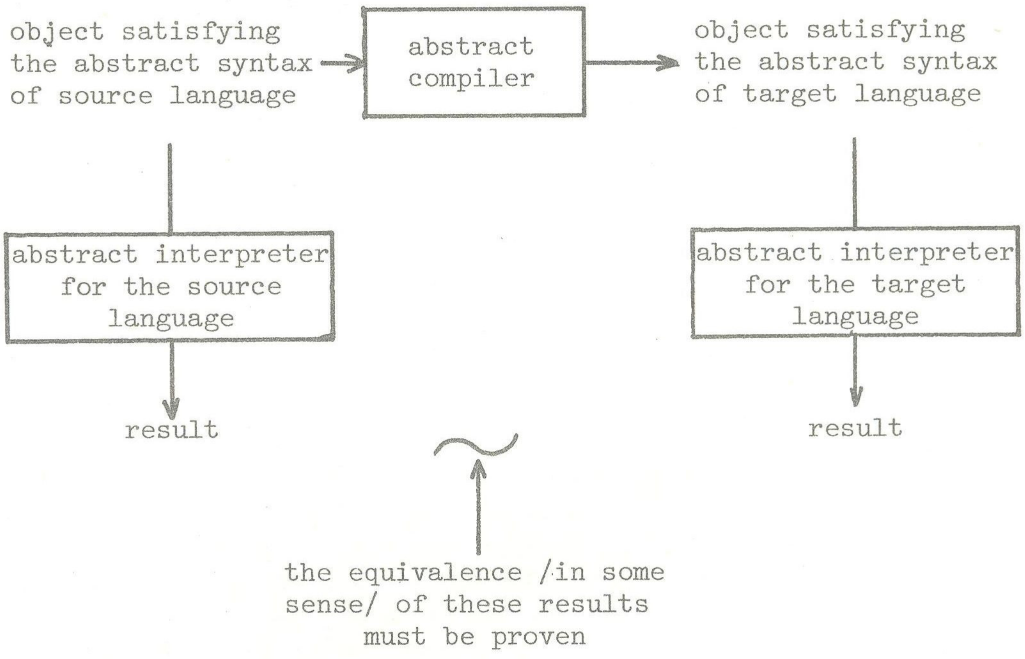

object satisfying the abstract syntax of source language
abstract compiler
object satisfying the abstract syntax of target language
abstract interpreter for the source language
abstract interpreter for the target language
result
result
~
the equivalence /in some sense/ of these results must be proven

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.

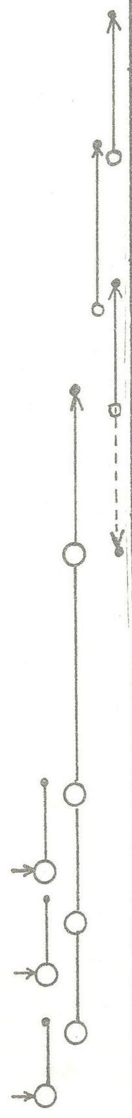
Since these activities are essentially independent they can be carried out and verified in parallel. You can verify formally (e.g. in the case of iii/ as described earlier), informally or by testing (e.g. in the case of i/ or ii/ if you have no better tools).

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 $\Box$ 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
$1\le f\le \mathrm{length}(A)$	$\underline{\mathrm{find}}(A,f)=\underline{\mathrm{reduce}}(A,f,1,\mathrm{length}(A))$	$(1\le p<f\le q\le \mathrm{length}(A)\supset\Box_p\le\Box_f\le\Box_q)\wedge\Box\sim A$
$P(A,f,m,n)\overset{d}{\equiv}$ $(1\le p<m\le r\le n<q\le \mathrm{length}(A)\supset A_p\le A_r\le A_q)$ $\wedge m\le f\le n$	$\underline{\mathrm{reduce}}(A,f,m,n)=$ $m<n\to\underline{\mathrm{reduce}}(\mathrm{vec}(x),f,\mathrm{lb}(x),\mathrm{ub}(x));$ $x:\underline{\mathrm{order}}(A,f,m,n)$ $T\to\underline{\mathrm{PASS}}:A$	$P(\Box,f,f,f)\wedge\Box\sim A$
$P(A,f,m,n)$	$\underline{\mathrm{order}}(A,f,m,n)=\underline{\mathrm{ord}}(A,f,m,m,n,n,A_f)$	$P(\mathrm{vec}(\Box),f,\mathrm{lb}(\Box),\mathrm{ub}(\Box))\wedge\mathrm{vec}(\Box)\sim A$
$R(A,f,m,g,h,n,s)\overset{d}{\equiv}$ $(m\le p<g\supset A_p\le s)$ $\wedge(h<q\le n\supset s\le A_q)$ $\wedge P(A,f,m,n)$	$\underline{\mathrm{ord}}(A,f,m,g,h,n,s)=$ $\underline{\mathrm{continue}}(B,f,m,i,j,n,s);$ $B:\underline{\mathrm{change}}(A,i,j);$ $i:\underline{\mathrm{up}}(A,g,s),$ $j:\underline{\mathrm{down}}(A,h,s)$	$P(\mathrm{vec}(\Box),f,\mathrm{lb}(\Box),\mathrm{ub}(\Box))\wedge\mathrm{vec}(\Box)\sim A$
$(i\le j\supset R(B,f,m,i+1,j-1,n,s))$ $\wedge(j<i\supset R(B,f,m,i,j,n,s))$	$\underline{\mathrm{continue}}(B,f,m,i,j,n,s)=$ $i\le j\to\underline{\mathrm{ord}}(B,f,m,i+1,j-1,n,s)$ $i\le f\to\underline{\mathrm{PASS}}:\mu_o(\langle\mathrm{vec}:B\rangle,\langle\mathrm{lb}:i\rangle,\langle\mathrm{ub}:n\rangle)$ $f\le j\to\mathrm{PASS}:\mu_o(\langle\mathrm{vec}:B\rangle,\langle\mathrm{lb}:m\rangle,\langle\mathrm{ub}:j\rangle)$ $T\to\mathrm{PASS}:\mu_o(\langle\mathrm{vec}:B\rangle,\langle\mathrm{lb}:f\rangle,\langle\mathrm{ub}:f\rangle)$	$P(\mathrm{vec}(\Box),f,\mathrm{lb}(\Box),\mathrm{ub}(\Box))\wedge\mathrm{vec}(\Box)\sim B$
$R(A,f,m,i,j,n,s)\wedge A_j\le s\le A_i$ $1\le i,j\le \mathrm{length}(A)$	$\underline{\mathrm{change}}(A,i,j)=$ $\mathrm{PASS}:\ \mu(A;\langle\mathrm{elem}(i):\mathrm{elem}(j,A)\rangle,\langle\mathrm{elem}(j):\mathrm{elem}(i,A)\rangle)$	$(i\le j\supset R(\Box,f,m,i+1,j-1,n,s))\wedge$ $(j<i\supset R(\Box,f,m,i,j,n,s))$ $\Box_i=A_j\wedge\Box_j=A_i\wedge(p\ne i\wedge p\ne j\supset\Box p=A_p)$
$R(A,f,m,g,h,n,s)$ $m\le p<g\supset A_p\le s$	$\underline{\mathrm{up}}(A,g,s)=$ $A_g<s\to\underline{\mathrm{up}}(A,g+1,s)$ $T\to\underline{\mathrm{PASS}}:g$	$R(A,f,m,\Box,h,n,s)\wedge s\le A_\Box$ $(m\le p<\Box\supset A_p\le s)\wedge s\le A_\Box$
$R(A,f,m,g,h,n,s)$ $h<p\le n\supset s\le A_p$	$\underline{\mathrm{down}}(A,h,s)=$ $s<A_h\to\underline{\mathrm{down}}(A,h-1,s)$ $T\to\underline{\mathrm{PASS}}:h$	$R(A,f,m,g,\Box,n,s)\wedge A_\Box\le s$ $(\Box<p\le n\supset s\le A_p)\wedge A_\Box\le s$

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. SZÁMKI 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. SZÁMKI 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.