PROGRAMMING AND COMPUTER SOFTWARE

A translation of *Programmirovanie*

January, 1986

Volume 11, Number 2

March-April, 1985

CONTENTS

The Russian press date (podpisano k pechati) of this issue was 3/18/1985. Publication therefore did not occur prior to this date, but must be assumed to have taken place reasonably soon thereafter.

The proof of this theorem can be obtained by using the method of proof of Theorem 1, the outline of the proof of Lemma 1 in [3], and the solvability of the problem of nonemptiness of a set perceived by a ΣTC-converter [2, 3].

LITERATURE CITED

1. L. P. Lisovik, "The problem of equivalence for converters over labeled trees," Dokl. Akad. Nauk Ukr. SSR, No. 6, 77-79 (1980).
2. L. P. Lisovik, "Metalinear recursive schemes over labeled trees," Programmirovanie, No. 5, 13-22 (1983).
3. L. P. Lisovik, "On the problem of equivalence for converters over Σ-trees with finitely reversible counters," Kibernetika, No. 5, 19-24 (1984).
4. L. P. Lisovik, "On solvable problems for metalinear schemes," Dokl. Akad. Nauk Ukr. SSR, No. 2, 130-133 (1979).
5. V. Yu. Romanovskii, "Solvability of problem of equivalence of linear unary recursive schemes with individual constants," Dokl. Akad. Nauk Ukr. SSR, No. 1, 72-75 (1980).
6. V. N. Red'ko and L. P. Lisovik, "The problem of equivalence for finitely reversible counters," Kibernetika, No. 4, 26-29 (1980).
7. S. Garland and D. Luckham, "Standard schemes, recursive schemes, and formal languages," Kibern. Sb., No. 13, 73-119 (1976).
8. E. L. Post, "A variant of a recursively unsolvable problem," Bull. Am. Math. Soc., 52, No. 4, 264-268 (1946).
9. V. M. Glushkov, "Theory of automata and formal transformations of microprograms," Kibernetika, No. 5, 1-9 (1965).

PROGRAMMING SOFTWARE AND SYSTEM PROGRAMMING

SIMPLE METHOD OF ORGANIZING A RELATIONAL DATA BASE PROVIDING INFORMATION ON THE STATE OF DEVELOPMENT OF LARGE PROGRAMMING SYSTEMS

UDC 51:681.3.06

N. N. Bezrukov

A method is proposed of entering information on the state of a project development into the user field of the table of contents of text libraries. The method was developed for the Unified Computer System operating system. Application of the operations of relational algebra to such a table of contents can provide quite comprehensive information as to the state of the project. The query language used for this purpose and the possibilities of service programs are described.

The use of software development tool systems has now become the principal method of increasing programming productivity and improving the quality of software reliability [1, 2]. It is not by coincidence that a powerful set of support tools is being developed in parallel with the development of an ADA translator [3].

It is well known that one of the main difficulties encountered in the design of large-scale systems is the organization of communication between individual designers. As noted by Brooks "... noncoordination with graphs, functional incompatibilities, and system errors are caused by the fact that the left hand does not know what the right hand does" [4, p. 61]. An important demand of the tools used in such project is that they be capable of recording information on the executed operations in a special data base, keeping track of the state of development (see, e.g., [5, chapter 19]). The availability of a data base (DB) and a proper query language increases the awareness of individual designers of changes taking place in the system that concern their special interests. It is important, for example, to be able to get answers to queries such as:

What modules have been changed in library A since May 10?

Translated from Programmirovanie, No. 2, pp. 44-53, March-April, 1985. Original article submitted December 2, 1983.

0361-7688/85/1102-009309.50

Which modules have been developed by programmer X?

When was module M lastly modified and by whom?

What is the difference between the latest version of module M and the former one?

The terminology used for this class of software tools is still not settled. The term most frequently used in foreign literature is programming environment. In [6] this term was translated "programming surroundings" and the abbreviation IPDF (integral program development facilities) has been proposed instead. Irrespective of which term will be generally accepted in the future, it is clear that powerful software tool systems founded on a special data base will become accessible to many programmers.

In the meanwhile, a simple data base on the state of system development can be organized in a rather improvised way. Here we describe a method which achieves the above aim (for the Unified System (US) operating system) at the cost of about one man-year by combining into a single software tool system already existing service programs after some local modification.

The main concept is to build in a relational data base [7] into the table of contents of every text library containing programs, documentation, or test data concerning the program product being developed. The table of contents is regarded as a relation containing information on the state of development of sections included in the library. Interpreting the table of contents of a library as a relation reflects the fact that the table of contents is physically a separate file whose entries are arranged in a lexicographically ascending order of section names. As is well known, the simplest implementation of relational data processing (see, e.g., [8-10]) is based on conventional sequential or index-sequential arrays.

Information on the state of development will be located in the user field of each item of the table of contents. This field (up to 62 bytes in size) is filled in the US operating system only for the loading module library. In text libraries the field can contain any information.

Structure of Section Ticket. The proposed user field structure (called the section ticket) reflects the experience of the author in the design and application of the NEAT software tool system [11]. The ticket includes the following nine components:

```
1 PASSWORD
2 A CHAR(12),/* AUTHOR — author's code*/
2 B CHAR(3),/*BIRTHDAY — date of creation (YYMMDD)*/
2 C CHAR(5),/*CORRECTED — date of correction */
2 D BIN FIXED,/ * DIMENSION — number of lines */
2 E BIN FIXED,/ * EDITION — number of version */
2 F BIT(32),/ * FILTER — modification filter */
2 G(5) CHAR(3),/ * GENERATIONS — generation stack */
2 H CHAR(1),/ * HOST-LANGUAGE — programming language */
2 I CHAR(18);/ * INFORMATION — designer's information */
```

As seen in the ticket structure, key words denoting fields have been selected in an alphabetic order (A — I). The field A always contains in the first four bytes the word NEAT which acts as an indicator of the described ticket format. The user has no access to these four bytes. The user can enter his code into the remaining eight bytes to facilitate the identification of the section "owner," thus providing some protection of the section text from unauthorized access.

The section creation date is recorded in field B as a packed unsigned decimal number (YYMMDD). The field C contains the correction data and the correction time (HHMM) in the first three and last two bytes respectively.

The number of lines in a section is recorded in field D and the serial number of the version, in field E. When the section is created, a one is entered into this field. Each next correction increments this number by one. Field F contains a filter whose purpose is to determine groups of designers who must be informed about modifications introduced into the given module.

The stack of disk addresses (in TTR format) of the last five versions is stored in field G. If the library was not compacted, the NEATED editor [12] makes it possible to read out any of these versions and so to recover the text of a section accidentally covered with other text. The field H contains a single-character code of the programming language (any language, X; PL/1, P; FORTRAN, F; Assembler, A) in which a given module is written. Finally, field I contains automatically entered information located between brackets in the first line of the section text, for example:

```
/* [generation of table of contents] */

C [calculation of trajectory]

* [minidisk driver]
```

Besides the components A through I, the following three "system defined" fields are also assumed to be specified: N (section name), T (TTR field), and M (section text).

The section ticket allows a completely new approach to the problem of access to information contained in one or more libraries. The now prevailing method of specifying a group of sections by listing their names can be replaced by defining a group of sections through conditions contained in the ticket data (e.g., by specifying the value of a certain field of the ticket).

The above aims can be achieved quite naturally by allowing certain operations of relational algebra (see, e.g., [6]) in the query language. Unfortunately, the comprehension of the concepts on which relational algebra is based is hampered by the terminology used in this field. To facilitate comprehension, we shall not use the "relational jargon" with its relations, domains, corteges, and other attributes.

Relational Operations on Tables of Contents. The principal operations traditionally associated with relational algebra are similar to the well-known set-theoretical operations but operate not on sets but on rectangular tables (relations). We shall consider only the following five operations: sampling (restriction operation in relational algebra), union, intersection, complementing, and difference.

The operation of sampling from a table of contents produces a new table of contents containing a subset of sections whose tickets satisfy the specified sampling predicate. For example, the request PROCLIB : B > 830101 makes it possible to select all items created after Jan. 1, 1983. The character ":" is used here as a symbol of unary operation and PROCLIB is the name of the corresponding DD map. The sampling predicate can be formed out of any number of comparisons with the connectives "&" (AND), "!" (OR) and "¬" (NOT) and allows the use of parentheses. An example of a more complicated request is

```
PROCLIB : A = 'Petrov' & C > 830101,
```

which specifies all sections created or corrected by programmer Petrov after Jan. 1, 1983. The operation of taking a subline ".." and indexing is allowed in the fields A, F, I, and N. For example, the request

```
PROCLIB : N[1..4] = 'NEAT'
```

selects from the table of contents all sections whose names begin with "NEAT."

Union, intersection, and difference are binary operations and produce a combined table of contents out of two starting ones. The union operation "+" of two tables of contents produces a table of contents containing all sections entering into the two original ones except duplicates in the second table. For example, the request

```
WORK + ARCHIVE
```

specifies information on the common part of the libraries WORK and ARCHIVE.

The intersection "/" produces a table of contents including all sections of the first table which are also included in the second table. For example, the request

```
WORK / ARCHIVE
```

makes it possible to find what sections of WORK are also stored in ARCHIVE. If one wants to check the identity not only of the name but also of certain other items of a ticket, the list of items should be given in special brackets "/[" and "]/" between the library names, for example,

WORK /[A, B, C]/ ARCHIVE.

A group of consecutive ticket items can be indicated by the symbol "..", i.e., A, B, C is equivalent to A..C. Considering this, the preceding request can be written as

WORK /[A..C]/ ARCHIVE.

The difference operation "—" produces a table of contents which includes all items of the first table for which there are no corresponding items in the second table. For example, the request WORK-ARCHIVE indicates what sections of WORK are not stored in the library called ARCHIVE. Any ticket items which must be included in the comparison must be indicated in separating brackets "—[" and "]—". For example, the request WORK —[A..I]— ARCHIVE makes it possible to find what sections of WORK are not found in ARCHIVE or have incompatible tickets.

The described operations can be combined as in conventional algebraic operations thus producing quite intricate requests. In practice most requests consist in sampling operations. Binary operations are only used infrequently.

ASSISTENT. The above operations make it possible to implement a service program for executing extensive operations on groups of sections. This program is named ASSISTENT.

The ASSISTENT operation must be controlled with the aid of a special language. Below we describe a query language NEATFACE developed for the NEAT software tool system and consisting of four types of operators: LET, MARK, RUN, and PUT. The syntax of these operators can be described in BNF as follows:

⟨operator LET⟩ ::=LET ⟨name of copy⟩ "=" ⟨expression⟩

⟨operator MARK⟩ ::= MARK ⟨name of field⟩ "=" ⟨literal⟩ FOR ⟨expression⟩

⟨operator RUN⟩ ::= RUN ⟨sample of START command⟩ FOR ⟨expression⟩

⟨operator PUT⟩ ::= PUT ⟨sample of section separator⟩ FOR ⟨expression⟩.

The LET operator creates a new table of contents by computing an expression consisting of the table of contents operators described above. For example,

LET TEMP = (WORK-ARCHIVE) : N[1..4] = 'NEAT' & H = 'P'.

The resulting table of contents acts as a "copy," i.e., is an independent sequential file (or library section) organized in the same way as the table of contents of conventional libraries of the US operating system (256-byte blocks with packed entries of variable length). It can be thus used in subsequent operations in the same way as regular tables of contents of libraries.

The MARK operator is used to correct the values of ticket fields for a given group of sections. All fields are treated as byte fields except the F field. The F field is assumed to be a bit field and the respective literal must be composed of ones and zeros only. The "*" symbol can be used as a literal and denotes that the former value of the given byte (bit) is to be preserved. The following are several examples.

```
MARK H = 'A' FOR PROCLIB : N[1..3]= 'ASM'

MARK F[9..16]='**1**100'
     FOR PROCLIB : C > 830501 & H= 'A'.
```

The symbol "*" in a literal denotes that the corresponding bit of field F is to remain unchanged.

The RUN operator is used to start a given procedure for the group of sections specified by the expression. For example, if the library SYS1.PROCLIB contains the procedure PCG with two parameters L (library name) and M (section name), the operator

```
RUN 'PCG L="NEAT.TEXTLIB",M=',N
     FOR TEXTLIB : C > 830501 & A='IVANOV'
```

will start this procedure for all sections in library NEAT.TEXTLIB corrected after May 1, 1983 and belonging to programmer Ivanov.

The PUT operator forms a sequential file out of all sections selected from a library isolating them from each other by a separator produced from a specimen indicated in the command, for example

```
PUT '*PROCESS(SIZE = 999999,ST,NT,A,X,N = ',N,');',M
    FOR  TEXTLIB : H = 'P' & C > 830501
PUT 'SECTION',N,'TTR = ',T,'AUTHOR: ',A,M
    FOR  TEXTLIB : H = 'P' & D > 256.
```

Informing Designers about Module Changes. Modifications of system modules can affect the performance of other modules being developed by other designers. Moreover, a programmer working on a given module may be unaware of all its applications and be unable to ensure re-editing all programs using this module.

To facilitate providing information on changes introduced in modules to designers we use the so-called "sphere of interest" method: the bits of field F are reserved for individual designers (or groups of designers). If a given module lies within the "sphere of interest" of a certain designer the respective bit in field F is set to one. A list of modules whose modifications concern a given designer can then be set up by simple sampling, for example

```
PUT 'SECTION TEXT MODIFIED', N, 'AUTHOR:', A, 'DATA',
C  FOR TEXTLIB  : C > 830501 & F[4] ='1'
```

(the 4-th bit of field F is assumed to be reserved for the given designer). Such automatic restriction of the flow of information concerning modifications is very useful in large projects in which the size of text libraries very much exceeds the capability of an individual designer to analyze it.

The field F can be corrected using the operation MARK described above. In addition, using tags it is possible temporarily to mark sections in which errors have been detected. Free bits of field F can be used for this purpose.

New Possibilities of Text Editor. The use of section tickets offers several new possibilities for the text editor. The most important of these are: provision of access to "archival" versions of a given section through the use of a stack of disk addresses (field G) of the former section generations, protection of section texts from unauthorized access, automatic consideration of the properties of edited text in the course of its correction.

The most practically valuable property is undoubtly the possibility of access to preceding generations of a given section. This makes it possible to avoid various minor but painful "microdisasters" when carelessness or unfavorable coincidence cause the destruction of a section whose text has been considerably changed since the time it was recorded on tape. Experience shows that programmers quickly master this facility and use it not only when a section text is destroyed but also to recover the original version of a module after the introduced modifications prove to be a failure.

Field A offers an opportunity to organize simple protection of the section text from unauthorized access: a new version can be recorded only by users whose code is identical with the contents of field A of the given section. This protection does not extend to sections whose ticket has blanks in field A: such sections are assumed to be accessible to all.

At present, both the creation and correction of program texts are executed with the aid of a display, frequently without a listing of the program version being corrected. The programming language code in field H provides an opportunity for automatic formatting of the corrected text while the editor is entered into the buffer allowing selective display of lines in accordance with the specified level of nesting. This facility is now implemented for the NEATED editor [12] making it possible to display a kind of "summary" containing only the headings and terminations of loops, procedures, etc., thus facilitating the detection and correction of a given class of errors. In principle, automatic determination of the programming language of the text being corrected makes it possible to consider the syntax in context search and replacement and also in certain editor commands (e.g., exchange of a group of lines), which is a step towards syntax-oriented editing of program texts [1, 2].

Implementation. Automatic writing into the user field in the design and correction of library sections requires modification of the text editor employed. The section ticket is

organized so that most of its fields can be filled out directly by the module that controls recording of a section in the library. The first step is thus modification of this module if the editor includes it. This method has been used in modifying the NEATED editor, and the NEABPAM recording module included in it can serve as an example.

If the employed editor has no special module for recording program sections in the library one can modify the macrocommands WRITE and STOW, calling auxiliary subroutines that generate the proper SVC. In the case of the WRITE macrocommand the subroutine must first analyze the first line of the text in order to find the value of field I. In the next calls the subroutine should increment the record counter by the number of logical entries in the block considering that the last block can be shortened. The counter can be stored in the 2-byte DCB field. The subroutine for the STOW macrocommand should generate a BLDL SVC to read the old value of TTR and the user field and then correct the ticket.

Implementing entry into the fields A, F, and H of the ticket requires more complicated editor modification and can be executed as a second step. Since most frequently used editors require indication of the user's name at start of the editing session, this information can be used for filling out field A. Fields F and H can be filled out by entering additional parameters into the instruction for writing the edited text into the section. Another version is to treat the section ticket as a line with number 0 which cannot be deleted or transposed but to which all correction commands can be applied.

To ensure access to previous generations of a section it is necessary not only to modify the module that generates the FIND macrocommand but also to expand the syntax of certain instructions. For the NEATED editor, for example, the syntax

⟨ command code ⟩"("⟨ name of section ⟩")"

has been changed to

⟨command code⟩"("⟨ name of section ⟩["—"⟨ number of generations ⟩]")"

For example, the command A(MEMB—2) reads the second generation of the MEMB sections, i.e., the text of the section as it was before the last two corrections, into the editor buffer.

To convert the existing text libraries (or rather to modify their tables of contents) we have developed the NEATFORM utility which converts the user field of each section into a specified format. In addition, several other utilities have been developed for generating a record of the table of contents of a library and decoding the section tickets, for printing out the library contents taking into account archival versions of sections, etc. A special operating mode of the NEATFORM utility is foreseen for clearing the G fields of each library section after compaction.

The ASSISTENT program is implemented with the aid of methods used in RYaOD and REGENT translators [8-10]. Unlike the latter, the program is an interpreter in which to each operator corresponds its own interpreting program. Since most requests are associated with the execution of sampling operations, the class of admissible predicates has been made expandable, allowing the user to define his own logical functions on the ticket elements. The user can, for example, write a PL/1 program which can determine if string X is contained in string Y

```
IS_IN:  PROC(X,Y)  RETURNS(BIT);
DCL (X,Y)  CHAR(*);
    RETURN (INDEX(X,Y) > 0);
END IS_IN;
```

This program should be translated and placed in the library of loading modules of the NEAT system. Then, to select from the PROCLIB library all sections whose field I contains the word "LENTA" one can use the expression

```
PROCLIB : IS_IN(I,'LENTA')
```

Operating Experience. The described program facilities are used at the KNIGA computing center from the middle of 1983. Besides being used in software development, the facilities have been successfully applied in practical education of students. It should be noted that the problems involved in student education at major universities are in a certain measure similar to problems encountered in the organization of large-scale programming system development. In particular, characteristic features of practical work of students are: persistent

lagging behind schedule, uneven use of computing capabilities (overloading at the end of a semester), and a large number of simultaneously debugged programs.

As is well known, simultaneous debugging of a program by a group of ten or more persons working with terminals connected to medium-power computers (ES 1022, 1033, 1040) drastically increases waiting time, which can exceed 10 min even if each of the debugged programs consists of not more than ten simple operators. This is associated with considerable loss of time on job planning and loading the various phases of the US operating system translators and linkage editor, as a result of which a program of the form

```
A: PROC OPTIONS(MAIN); I=1; PUT DATA (I); END A;
```

takes more than a minute to be translated and executed by an F translator with a resident operating system of the order of 200 K; the same program is processed by a debugging translator operating in the RUN mode in 30-40 sec (the section length being 120 K). In batch processing the translation speed can be increased by one-two orders of magnitude depending on the number of simultaneously processed programs. For example, a batch of 50 jobs of the given type is translated and executed by the debugging translator in 2-3 min (for the same section length).

It is thus possible significantly to increase the processing speed of a large number of similar jobs of the "translate–edit–compute," "translate–edit," "edit–execute," etc., type by analyzing the tables of contents of active text libraries and arranging batches for translation (editing) out of sections that have been modified up to this time and have appropriate tickets (fields H and, possibly, I). As a condition of including a section into such a batch one can use the correction time or, if "on the spot" updating is not used, the fact that the value of field T (TTR) exceeds a specified value (in conventional correction the library method of US operating system access transfers the section to the end of the library).

In the last case, the same method can be used for libraries that have no section tickets of the required format provided the library contains sections in a single programming language or the programming language can be found from the module name. The use of the ASSISTENT program for batch forming is not efficient since the operating speed and memory capacity are in this case much more important than the variety of executed functions.

The author has developed two service programs that efficiently solve the above problem. The NEATMAX program in a single call inspects the table of contents of a given library and writes into this library a section with a special name (e.g., ¤ MAX) having a single line only whose positions 73 through 80 contain the maximum value of the TTR field of the given library. Positions 1 through 72 of this line store a copy of the contents of the PARM field transfered to the NEATMAX program during the call. This field can subsequently be used as a separator of the batch elements.

After being called, the NEATRUN program inspects the table of contents of one or several libraries, including into the batch all sections for which the TTR field exceeds the value contained in the ¤ MAX section. If positions 1 through 72 of the single line of this section are not empty, the line is used as batch element separator. The symbol % of this line is then replaced with the name of the respective section. For PL/1 this line usually contains the *PROCESS map. At the end of inspection the NEATRUN program "on the spot" corrects the ¤ MAX section, writing the new maximum TTR value for the given library in the 73-80 field. This ensures continuous formation of a batch of jobs of the same type corrected up to the present time (since the preceding activation of NEATRUN).

The use of the above programs made it possible to relieve overloads taking place at the computing center at the end of each semester and to improve the center's efficiency. The programs confirm the advantages of applying the theory and methods of relational data bases in the development of service programs oriented on batch processing. A batch can consist not only of library sections but also of other structural units (entire libraries, files, or volumes).

* * *

Since the middle 1970s the design of software development tools using an integrated data base concerning the state of development of a project is given the same attention as the development of programming languages and translators in the early 1960s. The systems developed in the last decase can be divided into three groups depending on their capabilities and complexity [13-19].

The first group includes the most simple systems whose implementation requires several man-years. A data base on this level can be considered as lying half-way between the programmer and his tools, making it possible for him to fix a specified class of events out of all the events taking place in the system. An important advantage of such systems is their simplicity thanks to which they can be easily adapted to specific conditions. Moreover, such systems can be implemented by a single designer as proved in the present paper.

In systems belonging to the second group the data base has a more complicated structure and makes it possible to solve more complex problems [14-16]. The latter includes the problem of synchronizing of the source and object versions of library texts, automatic management of archives on tape, and verification of certain types of linkages between modules. An example of a modern system of such a level is the SDS system [14] operating under the control of the UNIX operating system and consisting of four main subsystems: processing of modification requests, control of program source texts, control of linkage compilation and edition, and verification control. The implementation of a commercial specimen of such a system takes several tens of man-years.

The third group has the most powerful data bases including information on relations between objects and events taking place in the system. While the first two groups are oriented on programming and debugging stages the last group is an attempt to cover the entire lifecycle of the product being developed [18, 19].

Such a data base is capable of providing answers to quite complex queries involving classification and selection of objects based on their properties, for example:

What modules are affected by modification of the structure of table T?

How many errors has programmer P made in module M?

By whom and when was the subsystem C tested?

Since the data base stores information both on all modules being developed and on all programmers, certain functions concerning passing information about specific events to all designers can be carried out automatically. For example, all programmers using module M can be automatically notified that an error has been found in the module and later that the error has been rectified.

Systems of this kind require hundreds of man-years to be fully developed and still are at an experimental stage [18, 19]. Nevertheless, it is even now obvious that a significant part of a programmer's daily work in translating, editing, and testing programs can be automated. Such systems can be regarded as some king of an "intelectual robot" serving the programmer which according to his instructions can carry out quite complex manipulations on the texts of interrelated programs and data [19].

The existence of several levels of software development tools does not mean that one level can completely displace others even if it obvious that the more complex tools will gradually gain advantage. This is in a certain measure similar to the situation existing in the field of programming languages where languages of the Assembler, FORTRAN, and PL/1 levels successfully coexist side by side.

LITERATURE CITED

1. A. I. Wasserman and S. Guts, "The future of programming," Commun. ACM, 25, No. 3, 196 (1982).
2. W. E. Howden, "Contemporary software development environments," Commun. ACM, 25, No. 5, 318 (1982).
3. Advanced Research Projects Agency, Requirement for ADA programming support environments ("Stoneman"), U.S. Department of Defense, Arlington, VA (1980).
4. F. Brooks, How to Design and Create Programming Systems [Russian translation], Nauka, Moscow (1979).
5. H. Mayers, Software Reliability [Russian translation], Mir, Moscow (1980).
6. A. P. Ershov, "Integral approach to current software problems," Kibernetika, No. 3, 11 (1983).
7. C. Deight, Introduction to Data Base Systems [Russian translation], Nauka, Moscow (1980).
8. A. A. Stognii, E. L. Yushchenko, V. I. Voitko, E. I. Mashbits, L. V. Vernik, and N. N. Bezrukov, "Man-machine data processing system oriented on nonprofessional users," in: Algorithms and Organization of the Solutions of Economical Problems [in Russian], Issue 14, Statistika, Moscow (1980), pp. 172-195.

9. N. N. Bezrukov, "Main-line optimization facilities in the REGENT relational report generator," Upr. Sistemy Mashiny, No. 6, 74 (1980).
10. N. N. Bezrukov, "Translation to a high-level language as a method of implementing problem-oriented languages based on relational algebra," Author's Abstract of Candidate's Dissertation, Kiev (1981).
11. N. N. Bezrukov, "Software development tools for PL/1 programmers," in: System Programming [in Russian], Kishinev State Univ., pp. 92-95.
12. N. N. Bezrukov, "Text editor with an extended system of commands for the US operating system," Programmirovanie, No. 3, 39 (1981).
13. B. W. Boehm, R. K. McClean, and D. B. Urfig, "Some experience with automated aids to the design of large-scale reliable software," IEEE Trans. Soft. Eng., SE-1, No. 1, 125 (1975).
14. P. White and M. R. Feay, "A software development system for reliable applications," IEEE Trans. Commun., 30, No. 6, 1363 (1982).
15. J. K. Cottrel and D. A. Workman, "GRASP: an interactive environment for software development and maintenance," Data Base, 11, No. 3, 584 (1980).
16. M. J. Rochking, "The source code control system," IEEE Trans. Soft. Eng., SE-1, No. 4, 364 (1975).
17. W. E. Riddle and R. E. Fairley, Software Development Tools, Springer-Verlag (1980).
18. H. Bratman, "The software factory," Computer, 8, No. 5, 28 (1975).
19. T. Winograd, "Breaking the complexity barrier again," SIGPLAN Not., 10, No. 1, 13 (1975).