PROGRAMMING AND COMPUTER SOFTWARE

A translation of *Programmirovanie*

May, 1989

Volume 14, Number 4 July-August, 1988

CONTENTS

The Russian press date (podpisano k pechati) of this issue was 7/31/1988. Publication therefore did not occur prior to this date, but must be assumed to have taken place reasonably soon thereafter.

HEURISTIC METHODS OF IMPROVING DISASSEMBLY QUALITY

N. N. Bezrukov

UDC 51:681.3.06

We consider methods of separating instructions and data, based on the creation of a "program map" – the collection of the descriptors of each byte. A disassembly method is proposed combining trace data and the comparison of memory dumps (the distinguished parts of the dumps correspond to variables). An implementation of the interactive disassembler NEADASM for ES/OS is described.

A system program that reconstructs the source code in assembly language from the load module is usually called a disassembler, and the reconstruction process is called disassembly. A disassembler is irreplaceable for analysis and modification of software when the source code is not available or has been lost. It is not by accident that most software development systems for microprocessors include disassemblers (see e.g., [1]). Another important field of application of disassemblers is debugging from memory dumps. A specialized disassembler version for this purpose may use compiling protocols of the system components for which the assembly-language source code is available to generate the values of the variables according to the type specified in the declaration statement (DC/DS for the ES computer assembly language). Disassembly of variable values allowing for variable types, i.e., a printout of values in decimal format for integer variables, in symbolic format for strings, in scientific format for floating-point variables, etc., will detect a whole range of problems "at a first glance" [2], essentially improving the productivity of debugging.

In terms of its ambiguity, the disassembly problem is very close to problems tackled by expert systems [3]: for any load module treated as a string of bytes, there exist infinitely many source codes that assemble into the same string of bytes. The differences between these codes may be more fundamental than just different variable names and labels. For instance, for the ES computer, the byte string with the hexadecimal representation

05EF D20B D0F0 1004

may be disassembled as

BALR 14,15 / MVC 240(12,13),4(1)

or as

BALR 14,15 / DC X'D20BD0' / LPR 0,4

The number of disassembly variants increases as the instruction set becomes more "saturated." In case of maximum saturation (e.g., with 256 single-byte instructions), any byte sequence following an unconditional jump instruction may be disassembled into instructions or data.

In the absence of additional information, correct separation of the load module into commands and data requires a high degree of mastery of the programming techniques for the particular computer and knowledge of the linkage organization between the subprograms and the structure of the control blocks in the particular OS. Even an expert may make mistakes, by mistaking some commands for data and vice versa. Therefore, the quality of disassembly is usually estimated by the degree of correct separation of commands and data.

Once commands and data have been correctly separated, disassembly quality can be further improved by reconstructing the types of the variables. This task should be considered as the first step in the next stage of restoring the program specification – the decompiling stage [4].

Historical Remarks. The history of disassembler development in the USSR [5-14] and elsewhere [15-19] hardly can be reconstructed because the relevant publications are widely

Translated from Programmirovanie, No. 4, pp. 81-93, July-August, 1988. Original article submitted December 17, 1986.

0361-7688/88/1404-0195$12.50 © 1989 Plenum Publishing Corporation

scattered in various collections and preprints and also because the research teams have largely worked in relative isolation (there are virtually no cross-citations of publications in this field, and some articles even do not contain a list of references). According to my data, the first Soviet disassemblers became available for the BESM-6 computer in the late 1960s [5, 6] and for the Minsk-32 computer in the early 1970s. Only batch disassemblers were available until the early 1980s. The first Soviet interactive disassembler for the ES computer was developed in mid-1980s [13], virtually simultaneously with the interactive disassembler for the K580 microprocessor [12] and the SM computer [14].

Disassembly Methods. Batch disassemblers, studied in [5-11, 15-19], are simple one- or two-pass programs (the second pass is used to place the labels) processing the load module or a memory dump. They are only capable of approximately separating the load module into commands and data. The problem of reconstructing the type of the data is usually not addressed at all. Various forms of the disassembly algorithm are traditionally used. This algorithm may be called "pseudoexecution" of the program being disassembled: starting at the beginning of the program or at some specified point, the disassembler makes an attempt to translate several current bytes into a command. If this cannot be done, the bytes are translated into data. The central data structure used for decision making in the so-called command directory. For most computers, this is a table with 256 entries, where each row contains the mnemonic of the instruction with the given operation code, its length in bytes, and some additional information.

In the absence of backtracking, the "pseudoexecution" algorithm often incorrectly disassembles some, possibly quite large, fragments of the program. An incorrect decision to translate the current bytes into a command or data is due to the limited context seen by the disassembler (the previous command and at best several following commands). These errors reduce the value of the results to such an extent that experienced system programmers often do not bother with disassembled code and prefer a simple hexadecimal printout of the load module with the choice of command codes printed at the bottom for each "suitable" (in ES computers, even) byte. In ES/OS, the utility IMASPZAP provides such a printout.

The complexity of disassembly tasks provides fertile ground for the application of heuristic methods, similar to those used for diagnosis in expert systems, such as MYCIN [3]. Note that elementary heuristics are utilized in numerous disassemblers. For instance, the popular batch disassembler RETRANS for ES/OS determines the value and the number of the base register by looking for commands of the form LR R1, 15 or BALR R1,0 in the initial section of the program fragment being analyzed and then accepts the register R1 as the base register. Unfortunately, no systematic analysis of heuristic techniques of improving the disassembly quality has been undertaken so far. In this paper, we will try to identify some classes of heuristics and suggests possible implementation techniques. Although the discussion is oriented to the ES computer, most of the heuristics can be extended to other processors (SM computers, K580, K1810, and others).

The Idea of Program (Descriptor) Map. the central idea for improving the quality of separation of commands and data is to create a descriptor for each byte of the program being disassembled. This descriptor accumulates useful information for the decision "command or data?". The collection of descriptors forms a program map (PM), which simplifies "Navigation in uncharted terrain." The PM enables us to separate the analysis of the load module code into well-defined passes, each of which analyzes the code from a certain viewpoint and stores the results in the descriptors of the corresponding bytes.

It is the presence of descriptors that enables us to unify in the process of disassembly the use of information obtained from a wide variety of sources, primarily from tracing or comparison of memory dumps. Although the idea of using trace data for disassembly is self-evident, it is difficult to implement without descriptors because of the need to process information from different sources simultaneously. Therefore, previous authors did not consider this possibility.

Tracing on one or several sets of input data in general will reconstruct only part of the commands and data comprising the program. Yet the traced fragments are unambiguously divided into commands and data. The presence of these fragments simplifies the disassembly of fragments not included in the trace. Moreover, since the trace produces the frequency and the range of values of each variable, it is easier to identify its type (integer, real, string, etc.).

Comparison of a number of dumps taken on different input data or in different phases of program execution in turn makes it possible to separate the domains that contain variables

from the domains that contain commands and constants (ignoring, of course, the pathological case of self-modifying programs). A sharper analysis will estimate the type of memory allocation to variables (static, automatic, or controlled allocation).

Formulation and Testing of Hypotheses in Disassembly. The program map provides a kind of testing ground for various hypotheses advances in the process of analyzing the load module code from various viewpoints. For instance, the hypothesis that byte K is the first byte of some command leads to a number of subsidiary hypotheses which should be tested in order to establish likelihood (in the sense of heuristic category [20]) of the original hypothesis:

all bytes from byte K to the first byte of an unconditional jump command are commands; here unconditional jump commands are the ES computer commands BC and BCR with mask 15, BAL, BALR (if R2 ≠ 0);

if byte K may be the beginning of a jump command and the jump address may be determined directly, then all the bytes starting with the jump address and until the nearest byte which may be the first byte of an unconditional jump command should be commands;

if byte K is the first byte of a memory access command and the operand address (A) may be determined directly, then the bytes starting with A, according to the operand size, should be data of the type corresponding to the type of the command (e.g., for the substraction command S, the data in memory should represent a binary number 4 bytes long);

if byte K is preceded by an unconditional jump command, then there should exist at least one conditional of unconditional jump command whose operand is the address of the byte K.

The NEADASM Disassembler. As part of the work on the NEAT software tool system [21], the author is developing the interactive disassembler NEADASM, which is currently used for testing various disassembly data structures and methods. In addition to the program map idea described above, the NEADASM disassembler employs various methods of analysis of program graphs and data flow graphs [22]. Moreover, there is a special emphasis on questions of conversational interaction with the human user in the process of disassembly, in particular, control of the output format on the monitor screen. Although the switch to the interactive mode substantially improves the quality of disassembly even when using the simplest algorithms of the "pseudoexecution" kind, it imposes a much greater load on the user. It is therefore necessary to develop an efficient organization of the man–machine interaction, similar to that used in most expert systems. NEADASM has taken only the very first steps in this direction by parametrizing the scope of the interaction on a number of logical scales, which are considered in what follows.

Data Structures for Disassembly. The data structures used in the simplest disassemblers are essentially the command directory and the label table. NEADASM, on the other hand, has eight data structures forming the disassembler data base. They include the program image (PI), the program map (PM), the list of linear groups of commands (LLGC), the relation IMPLIES, the relation CONTRADICTS, the command directory (CD), the bicommand directory (BD), and the data directory dictionary (DDD).

Program image (PI) is the load module or the dump read into main memory in unpacked form (two bytes in memory correspond to each byte of the load module). Since the minimum length of the command field in ES computers is half a byte, this representation simplifies various checks.

Program map (PM) is the central data structure of the disassembler. In ES computers, the commands are aligned at the half-word boundary, and therefore half-word descriptors (i.e., the descriptors of two consecutive bytes starting with an even address), and not byte descriptors, are used for memory reduction. Note that most microprocessors require byte descriptors, since they have one-byte commands. The descriptor structure has undergone major changes during NEADASM development. In order to get a feeling of these changes, it suffices to compare the structure described below with one of the early versions published in [23].

Each descriptor occupies four bytes (a word in ES terminology) and has the following structure:

```
1 DESCRIPTOR,
    2 OPTYPE BIT(2), /* COMMAND TYPE*/
    2 CONT BIT(3), /* SCALE OF CONTINUATION OF PREVIOUS COMMANDS */
```

```
2 LVAR BIT(3), /* SCALE OF JOINING VARIANTS OF LEFT COMMANDS */
2 PREV BIT(2), /* NUMBER OF PREVIOUS COMMANDS TO THE LEFT */
2 NEXT BIT(2), /* NUMBER OF FOLLOWING COMMANDS TO THE RIGHT */
2 XMARK BIT(2), /* OPERAND LIKELIHOOD SCORE */
2 BICOM BIT(2), /* PRESENCE OF BICOMMANDS TO RIGHT AND TO LEFT */
2 USE BIT(4), /* SCALE OF ACCESS TYPES TO A HALF-WORD */
2 EBCDIC BIT(4), /* NUMBER OF EBCDIC SYMBOLS TO LEFT AND TO RIGHT */
2 SYMTYPE BIT(4), /* SCALE OF SYMBOL TYPES */
2 BASEREG BIT(4); /* BASE REGISTER NUMBER */
```

The field OPTYPE contains the type of the possible command, as determined from the contents of the first byte of the half-word (00 – illegal code, 01 – ordinary operation, 10 – conditional jump, 11 – unconditional jump).

The field CONT determines the possibilities that the given half-word is a continuation of some command that began in one of the previous half-words (100 – the half-word may be a continuation of a 4-byte instruction, 010 – the second half-word of a 6-byte instruction, 001 – the third half-word of a 6-byte instruction).

The field LVAR contains information to the effect that the previous second, fourth, and sixth bytes contain command codes of length two, four, and six bytes respectively (i.e., "joining" variants with the previous commands to the left).

The fields PREV, NEXT give the number of commands before and after the current command (00 – none, 01 – one, 10 – two, 11 – three and more). The presence of commands to the left and to the right increases the likelihood of the hypothesis that the current half-word starts a command.

The field XMARK contains the likelihood score of the command operands on a 4-point scale (00 – no, 01 – more likely no than yes, 10 – more likely yes than no, 11 – yes). For instance, for the commands M, MR, D, and DR, the number of the first register should be even (when the register number is odd, the likelihood is zero). For RX commands (with the exception of LA) equality of the numbers of base and index register is unlikely (the score is 01). Similarly, for the commands CR and LR it is unlikely to have equality of the registers of the first and second operands, and for word-manipulation commands it is unlikely that the bias is not a multiple of 4 (for half-word manipulation commands, not a multiple of 2).

The field BICOM indicates if the pair of commands "previous–current" forms a frequently used combination (bicommand) in ES computers. The frequency of bicommands is estimated using a special table.

The field USE contains information on the usage of the current half-word in the program. The bits indicate that information is written into the given half-word (by the commands ST, STH, MVC, etc.), that the half-word is read as data (by the commands L, LH, A, AH, C, CH, etc.), that the half-word is read by the command EX, and that there is a jump command to the current half-word.

The field EBCDIC is filled if both bytes of the half-word contain codes corresponding to EBCDIC symbols from the standard 64-symbol set; it gives the total number of EBCDIC symbols surrounding the given half-word. This information is of considerable heuristic value, i.e., it formalizes the technique often applied by programmers in the analysis of dumps: the programmer uses the right-hand part of the dump (containing symbolic representation of the corresponding bytes) to identify a space with meaningful symbolic information and then attempts to interpret it as part of data. Note that the codes of the most frequently used commands in system programs for ES computers (with the exception of MVC, CLC, XC, and ST) are not reproducible by the standard 64-symbol set on line printers without lower-case letters. Therefore, a sequence of more than three EBCDIC symbols suggests the hypothesis that the corresponding memory space contains a text variable.

The SYMTYPE scale containing the types of the symbols in the given field is used to refine the field composition. Four different types of symbols are distinguished (capital letters, digits, separators, and Cyrillic letters that do not coincide in form with Latin letters).

The last field BASEREG gives the number of the base register used in the potential command. This field helps in the analysis of regions of constancy of the base register number, thus simplifying the placement of the directives USING and DROP. For commands whose format does not contain a base register this field is zero.

The information contained in descriptors may bring out a whole range of contradictions that arise when we advance the hypothesis that the current half-word starts a command. These contradictions, in particular, include the situations OPTYPE ≠ '0'B & CONT ≠ '0'B (the half-word cannot start a command and at the same time be a continuation of the previous command), PREV ≠ '0'B & NEXT = '0'B, or, conversely, the presence of more than a single 1 bit in XMARK, etc. The NEADASM user may specify a list of contradictions when the system should switch to interactive mode, types of contradictions when the hypothesis is rejected, and types of contradictions that should be ignored when evaluating the likelihood of the hypothesis that the current half-word starts a command. In this way, the man–machine dialogue can be restricted to cases when it is really necessary.

The next information structure is the list of linear groups of commands (LLGC). A linear group of commands (LGC) is a fragment of the load module having the property that if the first LGC byte starts a command, then all the other LGC bytes also contain commands. LGC clearly may contain only one unconditional jump (as the last command). Static program data may be located between two LGC or after the last LGC in the program. Therefore, in order to separate commands and data, it suffices to determine correctly the upper boundary of each LGC. For ES computers, there ar four main cases of data location: after the command BCR (this command usually serves to return to the calling program from a subprogram); after the command B (this command is often followed by local constants and parameters for SVC); after the command BALR or, more seldom, BAL (list of parameters); after the command SVC 13 (supervisor call for abnormal termination of the executing program).

A first approximation of the upper boundary of the LGC may be obtained by backtracking from the current command until we hit a descriptor with OPTYPE = '0'B & CONT = '0'B (data) or OPTYPE = '11'B (unconditional jump). In the latter case, we may assume that there is no data zone between the LGC examined. A potential symptom of a boundary between two LGC is the presence of a half-wrd with EBCDIC ≠ '0'B, i.e., the presence of a string of more than three symbols.

In order to refine the LGC boundary, we can attract all the sophisticated tools of analysis provided by descriptors. In particular, we can check the admissibility of bicommands at the join of two LGC. If the previous LGC ends with a subprogram call, then we should test the hypothesis that the data zone contains a list of parameters. A list of parameters always occupies a whole number of words, which may be interpreted as addresses, and often ends with a word with X'80' in the first byte (the end marker for a variable-length parameter list). If the previous LGC ends with the command SVC, then we should test the hypothesis that the command SVC is preceded by a parameter list. In ES/OS, each SVC has a fixed structure of parameter list in which the type of the elements is known and may be used for recognition. Moreover, if the previous LGC ends with an unconditional jump, then the first command in the next LGC shouod contain a label.

Command directory (CD) is a table in which every row contain s information about a certain command. A call to CD is via an intermediate vector of admissible commands 256 half-words long, which contains the indices of the corresponding CD rows for admissible command codes and zeros for inadmissible codes. The fields of each CD element include the length of the command, the address of the subprogram assessing the admissibility of the operands, and the command type (an 8-bit descriptor distinguishing between jump commands, word-manipulation commands, privileged commands, etc.).

Bicommand directory (BD) is a 256 × 256 byte table whose elements record the likelihood of occurrence of the particular two-command combinations (0 – "unrealistic," 1 – "conditionally unrealistic," 2 – "unlikely," 3 – "rare").

Unrealistic are two-command combinations the use of which in programs is meaningless (two comparison commands). Conditionally unrealistic are two-command combinations which are rendered meaningless by their operands (e.g., two identical commands with identical resulting operands are often meaningless, such as L – L, MVC–MVC, ST – ST, etc.). Unlikely are bicommands that do not occur in software of the particular class. Measurements of bicommand frequencies performed by the author suggest that system programs (we have used PRIMUS system

programs and programs of the SPEKTR data bank) use around 1,000 bicommands of the total of more than 30,000 (183^2) possible combinations. About half of those occur not more than once in every 10,000 commands are are classified as rare.

A likelihood level can be defined in NEADASM below which the corresponding bicommands are treated as inadmissible and the system switches to interactive mode for refinement. The default level is 3.

Data Directory Dictionary (DDD). The basic information structure intended for rconstruction of data types is the list of attributes (properties) of each variable, including the identifying attributes, representation attributes, usage attributes, and passport attributes. The identifying attributes include the address, the identifier (a short variable name not exceeding 8 symbols), and the name (a more accurate and full variable name, up to 32 symbols long). To improve readability, the user may specify the minimum frequency of a variable (MINFRQ) above which the disassembler substitutes the identifier for the full name, i.e., frequently used variables are represented by the abbreviated identifier and seldom used variables by their full name. Representation attributes include information on alignment, length in bytes, and type of the variable. Usage attributes include the frequency, the cross-references (the list of addresses and command codes in which the given variable is used), the cross-reference descriptor (an 8-bit field obtained by logical addition of the descriptors of the cross-referenced commands), access type (read, write, read—write, write—read), characteristic (local variable, input parameter, output parameter, internal variable, etc.), admissible range (filled manually or from trace results), and some other. Passport attributes include the version, date of creation, date of updating, and some additional information necessary in order to preserve the chronology of program disassembly, which is particularly important for large programs [24].

The structure of the DDD element makes it possible to derive some attributes from existing attributes. For instance, if a variable is only used by half-word manipulation commands, then it is most probably an integer variable or an attribute-type variable (if its range consists of two-three values, such as 0,1 or 1,2,3). If the variable is a read variable, then it is either a constant or an input parameter. The derivation can be automated by using a system of productions or rewrite rules which lead to a fairly natural program for recognition of patterns of this type [3] (some blocks of the MFÉYa compiler [25] are used, previously applied for recognition of syntactic constructs of high-level languages).

The DDD elements may be downloaded to a library. The external representation of the DDD element is compatible with the declaration of the variable in assembly language (DC/DS for variables, EQU for labels), for instance:

```
EOF DC F'0'                    ADDRESS 0160
*END-OF-FILE SYSIN             LOCAL VARIABLE
*END MARKER FOR FILE WITH DD-NAME SYSIN
*ADMISSIBLE RANGE: 0-1
*USED 3 TIMES
*REFERENCES* 006C - ST, 0078 - C, 0124 - ST
*TYPE: INTEGER
*ACCESS TYPE: WR
*TYPE OF COMMANDS: RX(F)
*DISASSEMBLY DATE: 18.06.86
*SECTION: XOPEN                VERSION 2
```

The representation in the form of DC/DS/EQU operators makes it possible to include these sections with the aid of the operator COPY when generating assembly language code, which reduces the size and improves the readability of disassembly results. The length of the comments can be controlled on a bit scale, in which each bit corresponds to an attribute of the DDD element.

Implementation Features. NEADASM was implemented following the general design strategy of syntax-driven processors. Note that there are no previous instances in the literature of such an approach to disassembler design as a variety of compiler. Our model partitions the disassembly process into five phases: initialization, scanning, parsing, semantic analysis, and source code generation. Each phase comprises one or several passes, always with an option to switch to interactive model.

Initialization phase includes passes that read the program to be disassembled or sections of this program into main memory, initialize the descriptors, and fill parts of the descrip-

tors from trace data or from comparison of memory dumps. The trace protocols are preprocessed by a special program, which outputs the information in a form suitable for direct filling of descriptors.

Scanning phase consists of passes that fill the PM and perform attribute induction of adjacent descriptors. Conflicts arising in attribute induction are resolved interactively.

Parsin phase identifies the LLGC, using the unambiguously classified half-words as reference points. Reference points containing commands are parsed to the potential LGC boundary (unconditional jump commands or unambiguously identified data). The parsing algorithm is a variant of the pseudoexecution algorithm. Parsin can be run backward from the reference point (backward pseudoexecution).

Semantic parsing phase is the most complex phase, including passes that identify the scope of the base registers, place the USING and DROP directives, fill the DDD, create the program schema, and perform its decomposition.

When determining the scope of the base registers, two lists are prepared: the list of registers used as the base register in jump commands and the list of registers used as base registers in data manipulation commands. For each element in this list, a table of addresses of the commands changing the values of the corresponding registers is constructed. Then in the interactive mode the user receives suggestions as to the recommended placing of the USING and DROP directives. To simplify the recognition of based structures (fictitious sections in the terminology of the ES assembler), the user may view on the screen a "cross-section" of the program which includes only the commands using the particular reister. Note that correct placement of the USING and DROP directives significantly influences the disassembly quality, and it is therefore advisable to develop supplementary utilities facilitating the solution of this problem.

Once the USING and DROP dirctives have been placed, the following passes fill the DDD. Unlike the existing interactive disassemblers, NEADASM recognizes the classical control structures (IF-THEN-ELSE, SELECT, CASE, the loops WHILE, UNTIL, FOR, N + $^1/_2$, etc.), which improves the understanding of the program being disassembled. Moreover, loop recognition is important for identifying common data types, such as arrays.

As a recognition method of control structures, NEADASM uses the apparatus of graph theory. In order to construct the program graph, each LGC is replaced with one input, one output command sequences (the so-called linear sections, LSEC). The LSEC adjacency list describes the graph of the disassembled program with linear sections as nodes (the program schema). Since not all the jump addresses have been identified in the preceding passes, the schema may be incomplete; however, in order to simplify the discussion, we assume that a complete schema is available.

The first stage in schema analysis involves decomposition into generalized control structures, the so-called hammocks [26-28]. A hammock is a subgraph of the program schema with one input and one output node. The hammock is connected with the rest of the schema only through these two nodes: all the arcs from the rest of the schema enter the input (inside) node, and all the arcs leaving the interior nodes of the hammock enter the output (outside) node. The output node is called an outside node because it does not belong to the hammock and it may also receive arcs from other nodes of the schema located outside the hammock. Several hammocks may share an output node. Sometimes the output node of a hammock is called focus.

Decomposition produces an ordered list of hammocks $H = \{h_1, h_2, h_3, \ldots, h_m\}$ which satisfies the following relationships: a) if $i \neq j$, then $h_i \neq h_j$; b) if $i < j$, then either h_i and j_j hve no common nodes, or all the nodes of h_i are contained in h_j. In other words, hammocks, like ordinary control structures, are either mutually independent or nested in one another. The list H includes all the standard control constructs (IF-THEN-ELSE, SELECT, CASE, the loops WHILE, UNTIL, FOR, N + $^1/_2$, etc.), since they all have one input and one output and are therefore hammocks.

Note that some authors (see, e.g., [27]) use a definition of hammock which prohibits the existence of an arc leaving the output (outside) node and entering the input (inside) node. From the viewpoint of completeness of decomposition, this restriction is unconstructive, since it loses the control structures enclosed in an UNTIL loop. For a more detailed discussion of these topics, and for an algorithm decomposing the program schema into hammocks, see [28].

In order to simplify the understanding of the program being disassembled, we classify the hammocks and precede the input LSEC of each hammock with comments, which contain the hammock type and the list of variables used in the hammock. For hammocks representing classical control constructs, the type is the construct name, and for all other constructs it is the number of nodes, their labels, and the index of the hammock schema in the catalog [28]. The list of variables may additionally give the type of access to the variable (read only, write only, read–write, and write–read), which simplifies hammock analysis. In general, in order to determine the access type, we have to apply methods of data flow analysis, but read-only and write-only access types can be determined by constructing for each LSEC in the hammock two logical scales, R and W: R(K) = 1 if the K-th variable is read and W(K) = 1 if this variable is altered in the particular section. Information about the entire hammock is obtained by logical addition of scales from the information about the separate LSECs.

* * *

Our studies have identified two particularly promising directions of improving disassembly quality.

The first direction is associated with increasing the level of the disassembled text by introducing backward compiling elements in the disassemblers [4]. Theoretically, disassembly is a particular case of the more general problem of backward compiling, i.e., reconstruction of code in a higher-level language from code in a lower-level language. Backward compiling is a new direction in compiling theory, which has not been studied so far. It involves a whole range of unsolved problems associated with recognition of data types, data structures (arrays, records, lists, etc.), and also standard instruction sequences (e.g., the procedure prologue and epilogue, extensions of system macro calls, etc.). LGC may be considered as a sentence in some context-sensitive language and it can be recognized by synthax-driven methods (the Floyd–Evans language [25] or attribute grammars).

The second, no less important, direction of improving the disassembly quality calls for application of algorithms and methods from expert systems. We should stress that the main tasks of disassembly (separation of commands and data, reconstruction of variable types and data structures) are particular cases of pattern recognition or, in a restricted sense, of program "understanding." We know that the understanding of speech is based on establishing some global context, which provides a framework for interpreting the spoken wrds and phrases. Therefore, a disassembly expert system should first establish "conceptual skeleton" of the program to be disassembled, by elucidating interactively the functions performed by the program and the corresponding data structurs. NEADASM implements only the simplest method of context tuning, which reduces to elimination of privileged commands from the command directory if it is known that the program is never executed in the "supervisor" mode, floating-point arithmetic commands if it is known that the program never performs numerical computations, etc. Particularly useful is the elimination of privileged commands, because data starting with X'80' often occur at the join of LGCs, and these data will be disassembled as the command SSM.

Disassembly expert systems may be classified as diagnostic systems. There is a certain analogy with other known systems, such as the "intelligent programming tutor" [29] and the "intelligent debugger" [30]. The development of disassemblers in the framework of artificial intelligence therefore appears to be most promising.

LITERATURE CITED

1. B. V. Antonov, S. F. Glazer, A. G. Malikov, and A. I. Shibalin, "An automated software design system for the one-chip microcomputer K1816VE48," Mikroprotsessornye Sredstva i Sistemy, No. 3, 25-27 (1986).
2. N. N. Bezrukov, "Some possibilities of listing processing in ES/OS," USiM, No. 5, 69-74 (1983).
3. M. Stefik, I. Eikins, R. Balzer, et al., "Expert system organization," Kiberneticheskii Sbornik, Novaya Seriya [Russian translations], No. 22, Mir, Moscow (1985), pp. 170-220.
4. N. N. Bezrukov, "Some issues in backward compiling theory," in: Issues of Improving the Functional Quality of Software in Real-Time Computing Systems [in Russian], KIIGA, Kiev (1987), pp. 10-22.
5. G. L. Maznyi, A Compiler from the Language of the BÉSM-6 Computer to Autocode [in Russian], Preprint No. 11-4950, JINR, Dubna (1970).

6. G. L. Maznyi, Description of a Compiler from the Language of the BÉSM-6 Computer to Autocode [in Russian], Preprint No. B2-11-4990, JINR, Dubna (1970).
7. V. V. Galaktionov, O. N. Lomidze, and G. L. Maznyi, " 'Strange' compilers in the Dubna monitor system for the BÉSM-6 computer," in: Conference on Programming and Mathematical Methods of Solution of Physical Problems [in Russian], Preprint No. D10-7707, JINR, Dubna (1974).
8. A. A. Bekasov and V. P. Bispen, "Disassembler organization for the K580IK80 microprocessor," in: Computational Processes and Structures [in Russian], No. 48, Leningrad. Inst. Aviats. Priborostroeniya, Leningrad (1981), pp. 22-24.
9. E. Yu. Mazepa and V. A. Sienko, A Decompiler from the Load Language to the ASS2 Language for the ES-1010 Computer [in Russian], Preprint No. N/2509, JINR, Dubna (1981).
10. A. N. Afanas'ev, V. N. Negoda, S. V. Skvortsov, and A. A. Smagin, "Software analysis tools for systems based on the INTEL 8080 microprocessor," in: Design and Application of Microprocessor Control Systems [in Russian], MIEM, Moscow (1984), pp. 217-220.
11. V. I. Yudkovskii and I. M. Aberemkov, "Reassembly of object modules of the SM computer," Prib. Sistemy Upr., No. 3, 9-11 (1985).
12. Methodological Materials and Documentation for Application Program Packages, No. 40, Micro-DDS – A Portable Operating System for Microcomputers, Part 2. Operator Manual [in Russian], MTsNTI-NIIPU, Moscow (1985).
13. A. A. Pavlenko, "An interactive disassembler for ES/OS," in: Issues of Improving the Functional Quality of Software for Real-Time Computing Systems [in Russian], KIIGA, Kiev (1987), pp. 51-57.
14. A. G. Grigor'ev, "An interactive disassembler for load modules in the RAFOS operating systems," Mikroprotsessornye Sredstva i Sistemy, No. 4, 19-23 (1987).
15. K. Datta, S. Bhattacharya, and G. Das, "Table-driven generalized disassembler for 8-bit microprocessors," Microprocessor Applications and Industrial Construction, Proc. Inf. Conf., Calcutta-New Delhi (1981), pp. 235-241.
16. R. H. Davis and A. J. Bathgote, "A Zilog Z8000 disassembler," Software Practice and Experience, 13, No. 11, 1055-1077 (1983).
17. W. Schardein, "Ein disassembler für den mikroprozessor Z80 auf der basis von APL," Elektron. Ind., No. 2, 28-30 (1982).
18. M. Antonova and É. Arakchiiski, "A reassembler for INTEL 8089," Avtomatika i Vychislit, Tekhnika (Bulgaria), 17, No. 5, 51-53 (1983).
19. P. Arno, "Mit etikett," Ct.-Mag. Comput. Techn., No. 11, 140-143 (1986).
20. D. Polya, Mathematics and Reasonable Deduction [Russian translation], Nauka, Moscow (1975).
21. N. N. Bezrukov, "Support tools for software development for automatic control systems," in: Applied Issues of Software and Hardware Reliability of Computing Systems [in Russian], KIIGA, Kiev (1985), pp. 23-30.
22. V. A. Evstigneev, Application of Graph Theory in Programming [in Russian], Nauka, Moscow (1985).
23. N. N. Bezrukov, "Improving disassembly quality by organizing the disassembler as an expert system," in: Issues of Evaluating the Functional Quality of Software in Real-Time Computing Systems [in Russian], KIIGA, Kiev (1986), pp. 16-24.
24. N. N. Bezrukov, "A simple method of organizing a relational database on the development status of large software systems," Programmirovanie, No. 2, 44-53 (1985).
25. N. N. Bezrukov, "A modification of the Floyd–Evans language," Programmironanie, No. 4, 53-64 (1979).
26. V. V. Martynyuk, "On analyzing the transition graph for an operator schema," Zh. Vychisl. Mat. Mat. Fiz., 5, No. 2, 293-310 (1965).
27. V. N. Kas'yanov, "Analysis of program structures," Kibernetika, No. 1, 48-61 (1980).
28. N. N. Bezrukov, "Decomposition of a directed graph into hammocks and its use for backward compiling of control structures," in: Theoretical and Applied Problems of Automatic Control Systems [in Russian], KIIGA, Kiev (1988), pp. 3-18.
29. D. R. Anderson and B. D. Reiser, "A Lisp tutor," in: Reality and Predictions of Artificial Intelligence [Russian translations], Mir, Moscow (1987), pp. 27-47.
30. W. L. Johnson and E. Soloway, "PROUST (An automatic debugger in Pascal)," in: Reality and Predictions of Artificial Intelligence [Russian translations], Mir, Moscow (1987), pp. 48-70.