Source Code Quality Classification Based On Software Metrics

Petr Vytovtov
Kalashnikov ISTU
Izhevsk, Russian Federation
osanwevpk@gmail.com

Evgeny Markov
Kalashnikov ISTU
Izhevsk, Russian Federation
zuper_cad@mail.ru

***Abstract*—Nowadays the software development speed is raising constantly. Therefore software development companies need a tool for checking source code quality to increase software maintainability and decreasing the number of errors in it. Moreover the systems of automated programming require the similar tool as well. As a result we have started developing a library for LLVM compiler which can evaluate source code quality at compile time and a programmer could receive information about source code quality and values of software metrics which are used for evaluating quality. In automated programming systems our library will be useful as well as a part of feedback step for increasing quality of generated source code.**

I. Introduction

There are a lot of published works about software quality and how to evaluate it. For example, in [1] Scott Pressman defines software quality as a conformity between software functionality and software requirements. This is a widespread approach to software quality. In ISO/IEC 25010 [2] software quality is defined as a combination of functionality, reliability, usability, efficiency, maintainability, and portability. This is another popular approach to defining software quality.

However it is possible to consider another way of evaluating software quality. It is based on source code metrics, when their values are used for calculating the quality value. This approach is developed by Thomas J. McCabe [3] (the cyclomatic complexity number is used for evaluating the software quality and testing complexity), Maurice Howard Halstead [4] (he has defined program level, volume and other measurement for evaluating software), and others. As a result of this approach there are several hybrid models (e. g. Maintainability Index or Cocol's metric) which will be considered below, but we think existing models do not allow to evaluate source code and software quality adequately which will be shown below.

Thus our purpose is to develop a new model for evaluating source code and software quality which will use values of basic source code metrics and implement it as a LLVM library. Also our model should combine two approaches described above, i. e. it should define software and source code quality in terms of ISO/IEC 25010 during the development process using source code measures.

This paper describes the LLVM IR language, the widespread basic source code metrics and which of them we chose, the hybrid metrics for evaluating software quality and why we think it is not enough to use them for evaluating software and source code quality, our approach to evaluating source code and software quality, the way to develop dynamic LLVM library, and the results of using our model.

The research presented in the article is based on our previous publications about choosing and evaluating basic and hybrid software source code metrics [5], [6].

II. LLVM IR introduction

The LLVM IR [7] is a way of representing source code in LLVM compiler before translating it into Assembler code, and it is possible to have an impact on source code with this representation at compile time (Fig. 1). This impact can contain optimization and evaluation procedures.

In the compilation process showed on the Fig.1 the source code is passed to CLang (a LLVM compiler frontend). Here CLang translates the source code to LLVM IR code and do optimization using this representation. Next, Clang passes the LLVM IR code to LLVM compiler which compile it to a binary file.

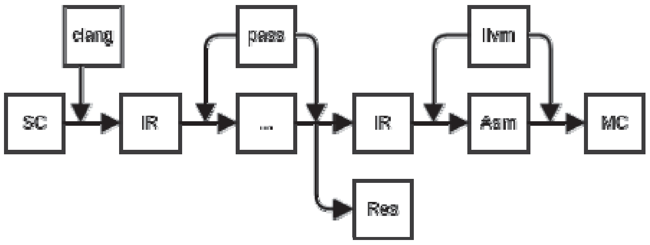

Fig. 1. The typical compilation process with LLVM compiler. *SC* – source code; *IR* – LLVM intermediate representation; *Asm* – representation with assembler language; *MC* – representation with machine code; *pass* – dynamic library for LLVM compiler; *Res* – the result of applying our approach.

The LLVM IR is a static single assignment hardware-independent low-level software source code representation. It is possible to use one optimization or evaluation tool for different high-level programming languages and hardware architectures with this representation. This is why we have chosen this way of source code representation for our analysis.

On the one hand the LLVM IR has modules and functions like high-level programming languages which is useful for splitting analysis from a whole program to small parts. On the other hand it uses low-level instructions representation which makes source code logic analysis easier.

In our approach we get source code evaluation result at compile time and save it and write into a separate file for the next analysis. This way of getting software parameters is described in Section VI.

Also we have considered two other ways of representing source code: an abstract syntax tree (AST) of high-level languages and Assembler language. In case of using AST the approach is limited with high-level source code representation. In case of using Assembler language the simple representation for analysis is but it is limited with hardware architecture.

III. Analyzing basic source code metrics

In this section we consider cyclomatic complexity [3], Halstead complexity measures [4], and some low-level source code metrics as a base for evaluating software and source code quality. The detailed review of software and source code measures is made earlier by Chernonozhkin S. K. [8] which we used in our measures analysis.

The process of evaluating software and source code quality is described in Section V.

A. Halstead complexity measures

Halstead described in his monograph 12 source code metrics: the number of distinct operators, the number of distinct operands, the total number of operators, the total number of operands, the program vocabulary, the program length, the calculated program length, the volume, the difficulty, the effort, the time required for program, the number of delivered bugs. We have analyzed these metrics for detecting which of them are useful for our approach. For this we have got 6 simple functions for calculating greatest common divisor written in C [9] (gcd1, gcd2, gcd3, gcd4, gcd5, and gcd6 respectively), translated them to LLVM IR, and compared values of Halstead complexity measures.

Table I. Basic Halstead measures for C-programs

	n1	N1	n2	N2	n	N	N’
gcd1	8	11	3	10	11	21	28
gcd2	5	7	2	7	7	14	13
gcd3	6	7	3	6	9	13	20
gcd4	4	4	2	5	6	9	10
gcd5	8	10	2	9	10	19	26
gcd6	20	48	6	45	26	93	101

Table II. Basic Halstead measures for LLVM-programs

	n1	N1	n2	N2	n	N	N’
gcd1	9	12	20	43	29	64	114
gcd2	9	24	21	51	30	75	120
gcd3	9	20	20	42	29	62	114
gcd4	9	17	19	37	28	54	109
gcd5	8	26	24	53	32	79	134
gcd6	13	88	76	190	89	278	522

First of all we have got the number of operators and operands for C (Table I) and LLVM IR (Table II) representations of test functions. Also we have removed operands which are used only twice (one write, one read) and calculated these metrics for cleared code (Table III).

Table III. Basic Halstead measures for cleared LLVM-programs

	n1	N1	n2	N2	n	N	N’
gcd1	9	21	6	19	15	40	44
gcd2	9	24	7	27	16	51	48
gcd3	9	20	6	18	15	38	44
gcd4	9	17	5	13	14	30	40
gcd5	8	26	3	17	11	43	28
gcd6	13	88	7	67	20	155	67

Here *n1* – the number of distinct operators, *N1* – the total number of operators, *n2* – the number of distinct operands, *N2* – the total number of operands, *n* – the program vocabulary, *N* – the program length, *N’* – the calculated program length.

Next we have calculated the Pearson correlation coefficients for showed datasets (Table IV). It is clear from Table IV that it is better to use a standard representation of C-code with LLVM IR than cleared representation. Also we have chosen only the number of operators and operands (n1, n2, N1, N2) and the calculated program length (N’) because the program vocabulary (n) and the program length (N) have linear dependence on the basic parameters of source code.

Table IV. The Pearson correlation coefficients between C-representation and LLVM-representation, and C-representation and cleared LLVM-representation

	n1	N1	n2	N2	n	N	N’
llvm	0.9	0.97	0.94	0.99	0.97	0.99	0.98
llvm-c	0.9	0.97	0.52	0.97	0.74	0.99	0.78

More complex Halstead measures use the program volume metric which has nonlinear dependence on the program length and the program vocabulary. Therefore we must test this measure for LLVM IR representation.

Table V. Volumes values for LLVM IR representations of test functions

	V	V*	V**
gcd1	310	11	15
gcd2	368	11	15
gcd3	301	11	15
gcd4	259	11	15
gcd5	395	11	15
gcd6	1800	11	15

Then, Halstead defines the volume (V), the potential volume (V*), and the bound volume (V**). Here the potential volume defines the average value of the program volume, and the bound volume defines the maximum value of the program volume. Therefore we calculated volumes for LLVM IR representations of our test functions (Table V). It is clear from the table that the program volume and its derivative measures are not useful for analyzing LLVM IR code.

B. The cyclomatic complexity

The cyclomatic complexity is the most widespread source code measure. Nowadays software engeneers use it everywhere for evaluating source code quality and complexity. But Thomas J. McCabe defined the cyclomatic complexity as a measure for evaluating the number of possible program executing paths. Therefore the cyclomatic complexity shows the complexity of testing process, not source quality.

Although you should not use the cyclomatic complexity as a measure of source code and software quality it is possible to use it as a part of a hybrid model of source code quality because the number of possible executing paths in program influences to the software quality but does not defines it. Therefore we have chosen the cyclomatic complexity and the number of nodes and edges of control flow graph for our model of source code quality.

C. Low-level source code metrics

Also we have chosen the volume of required RAM and the volume of the whole required memory (RAM + CPU memory) as parameters for our model. These values are helpful for evaluating source code quality. It is defined and proved in [5].

Thus we have chosen 10 measures as parameters for our source code quality model: the number of distinct operators, the number of distinct operands, the total number of operators, the total number of operands, the calculated program length, the cyclomatic complexity, the number of nodes in control flow graph, the number of edges in control flow graph, the volume of required RAM, and the volume of required memory at all.

IV. HYBRID SOFTWARE QUALITY METRICS

We mentioned above that the cyclomatic complexity is often used for evaluating software quality. In addition to it there are the Halstead program difficulty, the maintainability index (used in Microsoft Visual Studio), and Cocol's metric. Let's consider them in detail.

Halstead defines the program difficulty as quotient of the potential volume (1) and the real program volume (2) [4] where N – program length, n – program dictionary, n_2^* - program input/output-parameters. As mentioned above the measures which are derivative of the program volume cannot be used with LLVM IR representation. Also this measure is inversely proportional to the cyclomatic complexity (Fig. 2).

$$V^* = (2 + n_2^*)\log_2(2+n_2^*) \quad (1)$$

$$V = N\log_2 n \quad (2)$$

The maintainability index [10] is used in Microsoft Visual Studio for evaluating source code and software quality. But it is designed for calculating readability of source code because this measure uses the number of code lines, the cyclomatic complexity, and the Halstead Volume. These metrics show only one side of source code. Moreover the maintainability index is inversely proportional to the cyclomatic complexity for LLVM IR representation (Fig. 3). Therefore it is not possible to use it for our purposes.

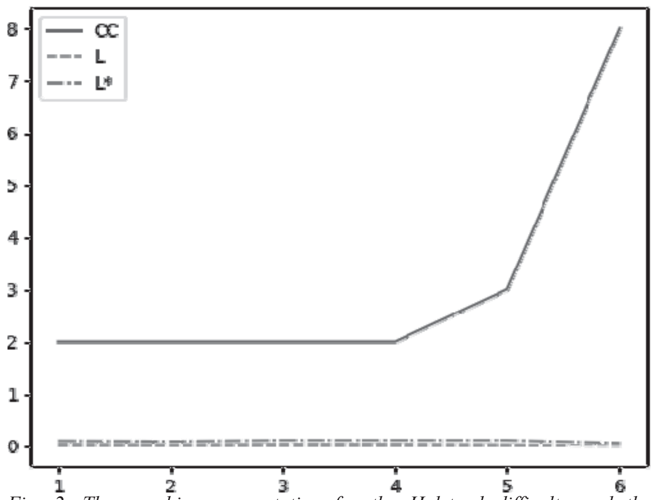

Fig. 2. The graphic representation for the Halstead difficulty and the cyclomatic complexity for test functions; L – the Halstead difficulty, *L** – the calculated Halstead difficulty; *CC* – the cyclomatic complexity.

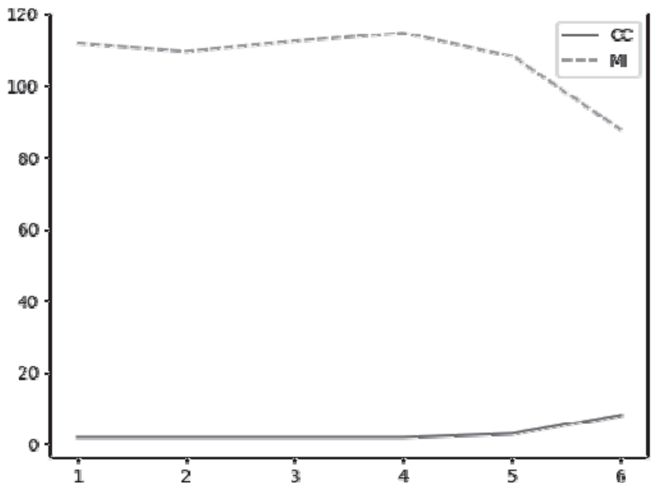

Fig. 3. The graphic representation for the maintainability index anf the cyclomatic complexity for test functions; CC – the cyclomatic complexity, MI – the maintainability index.

Also there are several models for Cocol's metric [11] where Cocol's metric is a normalized weight sum of basic source code metrics. For our purposes only the model of complexity is interesting. There are 6 input parameters in the model where the cyclomatic complexity is a base measure. But this model contains Halstead measures based on the program volume. The undesirability of using metrics of this type is shown above.

V. OUR APPROACH

In our approach we offer to combine terms of software quality from ISO/IEC 25010 and low-level source code static analysis and to use cluster analysis for classifying program functions to three groups: Good, Normal, Bad. Functions in Good group is well developed. Functions in Bad group must be optimized and improved. Functions in Normal group can be optimized and improved later.

We have decided to divide the process to two steps. During the first step we predict clusters for functions. During the second step we do the optimization of our model with experts' opinions.

TABLE VI. SOURCE CODE METRICS FOR TEST FUNCTIONS

	N1	n1	N2	n2	N'	n	e	CC	ram	All
1	21	8	43	20	110	4	4	2	96	385
2	24	8	51	21	116	4	4	2	64	481
3	20	8	42	20	110	4	4	2	96	385
4	17	9	37	19	109	4	4	2	64	385
5	26	7	53	24	129	7	8	3	64	450
6	88	12	190	76	517	21	27	8	160	1415

The Table VI contains values of source code metrics of our test functions [9] which we have defined above and calculated with out dynamic library for LLVM compiler presented in section VI. We have used these values in K-Means method (the first step) and got three clusters where functions 1, 3, and 4 are combined to one group, functions 2 and 5 combined to another group, and function 6 belongs to the third group. The cluster centers are presented in the Table VII.

If we start to compare the result of cluster analysis and terms of software quality from ISO/IEC 25010 (the second step) we will notice that test functions are divided according to representation of the software and source code quality.

TABLE VII. THE CLUSTER CENTERS FOR TEST FUNCTIONS

	Cluster 1	Cluster 2	Cluster 3
N1	19.33	25	88
n1	8.33	7.5	12
N2	40.67	52	190
n2	19.67	22.5	76
N'	109.67	122.5	517
n	4	5.5	21
e	4	6	27
CC	2	2.5	8
ram	85.33	64	160
All	385	465.5	1415

Table VII shows that we have high values of chosen measures for source code with low quality and their low values for source code with high quality.

Thus we can claim that our approach can be used in everyday software development tasks which is showed below.

VI. LLVM COMPILER DYNAMIC LIBRARY DEVELOPMENT

We have chosen LLVM compiler because its intermediate representation is simple to use in static analysis. The details of this representation is showed in Section II. All source code presented in this section is written in C++ using Qt framework.

LLVM compiler provides capability to use third-part dynamic library at compile time which is called Pass (Fig. 1). We have used this capability and created the dynamic library which calculates values of source code measures chosen above.

First of all we must create a skeleton of the library (Algorithm 1). It is necessary for registering our library in compiler. The source code analysis does in `runOnFunction` function. This function is called every time when the LLVM compiler processes new function in compiled program. There `F` is an object of function which is contains function instructions and arguments. There are similar functions for modules and basic blocks but we have focused on analyzing function objects.

In LLVM IR each function is divided to several basic blocks, and we should iterate through them and analyze control flow graph and instructions. We analyze control flow graph for calculating the cyclomatic complexity and the number of nodes and edges, and instructions for calculating Halstead measures.

Algorithm 1 The skeleton of the LLVM compiler dynamic library.

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/LegacyPassManager.h"
#include
"llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;

namespace {
    struct CQCPass : public FunctionPass {
        static char ID;
        CQCPass() : FunctionPass(ID) {}

        virtual bool
        runOnFunction(Function &F)
        { return false; }
    };
}

char CQCPass::ID = 0;

static void
registerSkeletonPass(const PassManagerBuilder &,
                     legacy::PassManagerBase &PM)
{ PM.add(new CQCPass()); }

static RegisterStandardPasses
RegisterMyPass(
        PassManagerBuilder::EP_EarlyAsPossible,
        registerSkeletonPass);
```

For each basic block we can get its connections to other basic blocks and build control flow graph (Algorithm 2).

After this we can use simple methods for getting three function parameters: the cyclomatic complexity, the number of nodes and edges of the function control flow graph.

Algorithm 2 The process of building the control flow graph.

```
++m_basicBlocksCounter;
if (basicBlock.getNumUses() != 0) {
    for (User *user : basicBlock.users()) {
        QPoint edge((size_t)user,
                    (size_t)&basicBlock);
        if (!m_edges.contains(edge))
            m_edges.append(edge);
    }
}
```

Algorithm 3 Methods for checking operators and operands in functions.

```
void
FeaturesParser::checkOperand(Value *operand) {
    if (operand->getType()->getTypeID() ==
            Type::VoidTyID)
        return;
    if (m_operands.contains(operand))
        ++m_operands[operand];
    else
        m_operands[operand] = 1;
}

void
FeaturesParser::checkOperator(QString opcode) {
    if (m_operators.contains(opcode))
        ++m_operators[opcode];
    else
        m_operators[opcode] = 1;
}
```

Algorithm 4 Method for calculating used memory volume for operands.

```
unsigned
FeaturesParser::checkType(Value *value) {
    Type *type = value->getType();
    switch (type->getTypeID()) {
    case Type::HalfTyID:
        return 16;
    case Type::FloatTyID:
        return 32;
    case Type::DoubleTyID:
        return 64;
    case Type::X86_FP80TyID:
        return 80;
    case Type::FP128TyID:
    case Type::PPC_FP128TyID:
        return 128;
    case Type::IntegerTyID: {
        IntegerType *integerType =
                dyn_cast<IntegerType>(type);
        return integerType->getBitWidth();
    }
    case Type::PointerTyID:
    case Type::VoidTyID:
    case Type::LabelTyID:
    default:
        return 0;
    }
}
```

For getting values of Halstead measures we check each instruction and its return value and operands. For this purpose we have two functions (Algorithm 3).

The last what we need is information about memory. For getting it we use the function based on standard values of used memory with datatypes (Algorithm 4).

Next we write got values into csv-file which we can use for following cluster analysis.

VII. Experimental results

In our experiments we have took Mozilla Firefox browser source code which contains 1349921 functions. We have analyzed them and split to three classes with K-Means method (Table VIII). We have got 1349321 function in the first group, 284 in the second group, and 316 in the third one. Next we have done the same for LLVM compiler source code which contains 868436 functions and got 868415 functions in the first group, 4 functions in the second group, and 17 functions in the third one. The clusters centers is presented in Table IX. Some huge values in these table is a result of few functions in groups.

TABLE VIII. The cluster centers for Mozilla Firefox browser functions

	Cluster 1	Cluster 2	Cluster 3
N1	23.92	15.97	3060.55
n1	6.84	7.18	17.94
N2	55.55	23.03	7905.95
n2	25.13	16.12	2679.99
N'	163.11	85.36	31475
n	3.46	6.29	332.82
e	3.22	3.29	477.4
CC	1.76	4.29497e+09	146.57
ram	89.23	87.89	3969.32
All	222.5	88.47	30091.33

TABLE IX. The cluster centers for LLVM compiler functions

	Cluster 1	Cluster 2	Cluster 3
N1	20.21	17817	13682.36
n1	6.83	15.25	13.94
N2	51.1	51773.5	39063.06
n2	25.98	17583	13184.77
N'	161.76	248190	182112.88
n	2.5	723.25	2174.88
e	2.01	1226	3719.77
CC	1.52	504.75	1546.88
ram	80	6072	3072
All	160.96	562696.25	215996.94

Next we calculated the Pearson correlation coefficients between cluster centers of functions from Mozilla Firefox browser and functions from LLVM compiler (Table X).

From this result we can predict that it is possible to classify functions from LLVM compiler with cluster centers from Mozilla Firefox browser and vise versa. For LLVM compiler functions we have got 868292 functions in the first group, 0 functions in the second group, and 144 functions in the third one. For Mozilla Firefox functions we have got

1349628 functions in the first group, 1 function in the second group, and 292 function in the third one.

TABLE X. THE PEARSON CORRELATION COEFFICIENTS BETWEEN CLUSTERS FOR MOZILLA FIREFOX BROWSER AND LLVM COMPILER

		Mozilla Firefox functions		
		Cluster 1	**Cluster 2**	**Cluster 3**
LLVM functions	**Cluster 1**	0.98	-0.28	0.96
	Cluster 2	0.92	-0.17	0.9
	Cluster 3	0.94	-0.20	0.99

Next we have combined the data for both projects and calculated cluster centers for their functions (Table XI) and the Pearson correlation coefficients between them (Table XII). From 2218351 function we have 2218049 functions in the first group, 284 function in the second group, and 24 functions in the third one.

TABLE XI. THE CLUSTER CENTERS FOR BOTH PROJECTS FUNCTIONS

	Cluster 1	**Cluster 2**	**Cluster 3**
N1	22.88	15.97	14342.79
n1	6.84	7.18	15.08
N2	54.89	23.03	40012.79
n2	25.83	16.12	13597.54
N’	166.85	85.36	18824.64
n	3.13	6.29	1729.79
e	2.82	3.29	2896.71
CC	1.69	4.29497e+09	1168.92
ram	86.16	87.89	4349.33
All	202.24	88.47	285890.5

TABLE XII. THE PEARSON CORRELATION COEFFICIENTS BETWEEN CLUSTERS FOR MOZILLA FIREFOX BROWSER, LLVM COMPILER, AND BOTH OF THEM

		All functions		
		Cluster 1	**Cluster 2**	**Cluster 3**
LLVM functions	**Cluster 1**	0.99	-0.28	0.91
	Cluster 2	0.94	-0.2	0.99
	Cluster 3	0.9	-0.17	0.98
Mozilla Firefox functions	**Cluster 1**	1	-0.27	0.94
	Cluster 2	-0.27	1	-0.19
	Cluster 3	0.96	-0.23	0.97

The values calculated during cross analysis are similar to initial result of cluster analysis and we can affirm that our approach is right for purposes of software and source code quality classification and can be used in daily software development. But it needs some improvements.

However we cannot determine exactly which source code quality group each cluster shows. We can make only assumptions. In the further work we will do expert survey for solving the problem.

VIII. FURTHER WORK

In the previous section we have described the experiment of analyzing source code of two big open source projects and showed that our approach can be used for analyzing software and source code quality. However we highlighted that there are some problems with it.

Firstly, in Table VIII and Table IX we can see several huge values. As a result in Table X some correlation coefficients may be disputable and they require clarification. It is a problem of few functions in groups 2 and 3. Therefore we should increase the number of analyzed projects and functions for fixing it.

Secondly, we cannot affirm that our approach is right without expert survey. It is necessary because there is no universal definition of quantification of software and source code quality and we should average opinions on it from different experts.

Thirdly, the expert survey can show us that we need to add and (or) remove some metrics from our model. The first what we will do is combining the model presented in this paper and the results of our previous model for evaluating functions time complexity [6]. It can help us to improve software and source code quality evaluating.

Thus our next work will be focused to increasing the number of analyzed software, adding the time complexity model to the model described in this paper, making the expert survey for clarification, and combining processes of calculating metrics values and evaluating source code quality.

IX. CONCLUSION

In this paper we have described our approach for evaluating software quality with its source code and its integration into compiling process with LLVM compiler and presented the model and tool for it.

The experimental results have showed that our research direction is right but the approach needs some improvements which is defined in Section VIII.

Thus we have a working method for evaluating software and source code quality which needs small improvements. The source code of our dynamic library for LLVM compiler is licensed under terms of GNU GPLv3 and free available on GitHub [12].

REFERENCES

[1] Pressman, Scott. Software Engineering: A Practitioner's Approach (Sixth, International ed.), McGraw-Hill Education Pressman, 2005.

[2] ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, Web: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733.

[3] Thomas J. McCabe, “A Complexity Measure”, IEEE Transactions on Software Engineering, Dec. 1976, pp. 308–320.

[4] Halstead, Maurice H. Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc., 1977.

[5] Vytovtov P. K., Markov E. M., M. Aiman Al Akkad, “Analysis of Software Code Metrics for Defining Their Priority for Cocol's Metric”, Instrumentation Engineering in the XXI Century. Integration of Science, Education and Production, Nov. 2014, pp. 543-546.

[6] Vytovtov P. K., Markov E. M., “Neural Network Development for Classifying Algorithms by Time Complexity”, Young researches –

the acceleration of scientific and technological progress in the XXI century, Apr. 2016, pp. 872-876.

[7] LLVM Language Reference Manual, Web: http://llvm.org/docs/LangRef.html.

[8] Chernonozhkin S. K., The support methods and tools for high-quality software development, 1998.

[9] test-functions-for-halstead-complexity-measures.c – GitHub Gist, Web:https://gist.github.com/osanwe/2c4e958d088617f4cc3182a505f8cb58.

[10] Code Metrics – Maintainability Index, Web: https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/.

[11] S. V. Swezdin, "Problems of measurement of code quality", Bulletin of the South Ural State University. Series: Computer technologies, automatic control, radio electronics, №2 (178) / 2010, pp. 62-66.

[12] osanwe/CQCPass – GitHub, Web: https://github.com/osanwe/CQCPass.