A New Tool for Calculation of a New Source Code Metric

Emil Stankov, Mile Jovanov, Kjiro Gjorgjiev and Ana Madevska Bogdanova
Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University
Skopje, Macedonia
{emil.stankov, mile.jovanov, ana.madevska.bogdanova}@finki.ukim.mk, kire9dk@gmail.com

Abstract—**Teaching programming is an activity that becomes more and more popular. Assessment of the students that attend introductory courses in programming can partly be done through presentation of simple source code fragments to them. Students should be able to provide the answer to the question: "What is the output of the given code?" When preparing the code segments, teachers should be aware of the complexity ('weight') of the code, and should also try to provide same or similar complexity tasks for all students. Nowadays, when there is a lot of research on the issue of automatic question production, the necessity of having a way to automatically measure the weight of some code is indisputable.**

In this paper we present a new source code metric that helps defining the weight of the code and a new tool that employs it.

Keywords—source code analysis; source code weight; teaching programming

I. Introduction

Teaching programming is an activity that is becoming more and more popular. Undoubtedly, this is due to the popularity of computer science nowadays, and moreover of programming, as its essential part. One of the most important challenges that teaching programming brings, particularly in courses attended by large numbers of students, is assessment.

Assessment of students' work, as a significant and almost inevitable part of the process of education on high school and university level, can be implemented in different ways, depending on the type of knowledge (theoretical, practical, or both) that is expected to be gained by the students. When it comes to programming courses, especially introductory ones, assessment can partly be done by presenting simple source code fragments to students and determining their basic level of knowledge and understanding of the programming language in which the respective fragments have been written. One clear means to do this is by asking them questions of the form: "What is the output of the given code?" Although these types of questions can hardly assess capabilities such as problem solving, algorithmic thinking or deep logical reasoning, they can (at least) give a good insight into the understanding of the basic programming constructs of the underlying programming language, as well as the comprehension of some basic programming concepts in general. According to the Bloom's taxonomy of educational objectives [1], comprehension is a stage that precedes application, so it is a good practice to examine the students' ability for comprehension, before asking them to apply their knowledge.

In order to achieve objective and fair assessment on a particular course exam, all students taking the exam must be asked questions of same or very similar complexity, i.e. questions that require the same level of knowledge to provide a correct answer. In the context of teaching programming and the types of questions mentioned previously, this means that teachers should be aware of the program codes complexity when preparing these questions, and should always try to provide questions containing same or similar complexity code fragments for all the students. As noted previously, this becomes particularly challenging when working with large groups of students, since teachers must maintain the consistency in creating the required large number of "same complexity" questions for the students' tests.

Programming is a compulsory subject in every computer science educational curriculum, and thus, usually lots of computer science students enroll in these courses. This means that vast majority of programming teachers have to deal with the problem of consistency of question complexity mentioned above. A possible solution to this problem is to provide a means of automatic production of questions. In the past ten years there has been a significant research on the issue of automatic production of questions of good quality for educative assessment needs.

Our wider research is concentrated on the automatic production of questions containing a source code or a chunk of a source code. In order to achieve the desired complexity consistency in the process of automatic production of questions for programming courses, we must have a way to automatically measure the complexity ('weight') of source codes.

In this paper we present a source code metric acceptable for the previously mentioned purpose (calculation of the complexity of a given code), and a new tool that employs the metric to produce new source codes.

The paper proceeds as follows. In Section II, some common software metrics typically used to measure the complexity of a given source code and their drawbacks are described. In Section III we present our new proposed metric. Section IV presents the tool, and Section V gives the results of a case study of the tool. The conclusion and remarks on the future work are given in Section VI.

II. Related Work

In this section we will describe some common software metrics typically used to measure the complexity of a given source code. The examples will include the Halstead Complexity [2] and the Cyclomatic Complexity (McCabe's Complexity) [3]. We will discuss the advantages of using each of these metrics, as well as their respective drawbacks, and we will consider the possibilities for their application in the domain of teaching programming.

A. The Halstead Complexity

The Halstead complexity metrics [2] are among the oldest measures of source code complexity. They were introduced in 1977 by Maurice Halstead, as a principal attempt to quantitatively estimate the effort of the programmer when writing a particular program's source code. The goal of Halstead's research at that time was to identify measurable properties of software, as well as to establish the relations between them.

Halstead interprets the source code of a given program as a sequence of tokens, and classifies each of the tokens as an operator (traditional operator such as '+', '*', or '>', statement separator such as ';', or a keyword such as 'if', 'return' or 'continue') or an operand (literal expression, constant or variable). The four basic metrics defined by Halstead are the following:

- n_1 – number of unique (distinct) operators
- n_2 – number of unique (distinct) operands
- N_1 – total number of operators
- N_2 – total number of operands

All the basic metrics are calculated by counting the frequencies of each of the operator and operand tokens in the program's source code. The other Halstead metrics are derived from them as explained below:

- Program vocabulary (n) – it is defined as the sum of the number of distinct operators and the number of distinct operands, i.e. $n = n_1 + n_2$.
- Program length (N) – it is defined as the sum of the total number of operators and the total number of operands, i.e. $N = N_1 + N_2$.
- Calculated program length ($\hat{N}$) – this metric provides a way of measuring the relationship between the program length N and the program vocabulary n. It is given by $\hat{N} = n_1 \cdot \log_2(n_1) + n_2 \cdot \log_2(n_2)$.
- Program volume (V) – this metric describes the size of the implementation of a given algorithm, expressed in mathematical bits. It can be calculated as the program length times the logarithm (base 2) of the size of the program vocabulary, i.e. $V = N \cdot \log_2(n)$.
- Difficulty (D) – this metric is also known as error proneness. According to Halstead, the level of difficulty of a program (or its error proneness) is proportional to the number of unique operators, as well as to the ratio between the total number of operands and the number of unique operands, i.e. $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$. This means that if we use the same operand(s) many times in our program, it will be more prone to errors. The metric also suggests that sources of program difficulty are repetition of operands and introduction of new operators in the program.
- Program level (L) – it is defined as the inverse of the difficulty level of the program, i.e. $L = \frac{1}{D}$. This means that a high level program is less prone to errors than a low level program.
- Effort (E) – this metric refers to the effort required to implement or to understand a program. Halstead suggests that the effort is proportional to the level of difficulty and the volume of the program: $E = V \cdot D$.
- Time (T) – it refers to the actual coding time, i.e. the time required to implement or to understand a program, expressed in seconds. As expected, this time is proportional to the effort required to write the program. Halstead has experimentally found that a good approximation for the time can be obtained by dividing the effort by 18 ($T = \frac{E}{18}$), but further experiments may be conducted to calibrate the measure.

Some of the advantages of the Halstead's complexity metrics are the facts that they are simple to calculate, applicable to any programming language, and that they do not require in-depth analysis of the program's structure. These metrics measure the overall quality of programs and can predict the maintenance effort. Many studies support the use of the Halstead's metrics in prediction of the programming effort, as well as of the number of programming errors.

B. The McCabe's Complexity

The McCabe's complexity [3] (also known as cyclomatic complexity or program complexity) is one of the most widely accepted software metrics, and undoubtedly – the most widely used static software metric. It was developed in 1976 by Thomas J. McCabe. This metric measures the number of linearly independent execution paths through a program's source code. For example, if the source code under consideration does not contain decision points (such as if statements or for/while loops), its complexity will be 1, since there exists only a single path through this code. On the other hand, if the source code contains a single decision point, then there will be two paths through the code: one path where the condition corresponding to the decision point evaluates as logically true and the other one where the condition evaluates as logically false.

Formally, the cyclomatic complexity (M) of a structured program is defined by $M = E - N + 2 \cdot P$, where E is the number of edges, N is the number of vertices and P is the number of connected components in the control flow graph of the program. For a single program, P always equals 1, so the formula becomes $M = E - N + 2$.

McCabe proved that the cyclomatic complexity of any structured program with a single entrance point and a single exit point is equal to the number of decision points plus one. However, we must note that this applies only to decision points on the lowest level (machine-level instructions). When writing programs in high-level languages, programmers often use compound conditions and these decision points should be counted in terms of the predicate variables involved in the compound condition (for example, 'if condition1 and condition2 then …' should be counted as two decision points, since it is equivalent to 'if condition1 then if condition2 then …' at machine level). For programs with more than one exit point, the cyclomatic complexity can be calculated as $d - e + 2$, where d is the number of decision points and e is the number of exit points.

One of the most important advantages of the cyclomatic complexity metric is that it can be used to guide the process of dynamic testing of the functionality of the programs (using test cases). Because the cyclomatic number describes the control flow complexity, it is clear that programs with high cyclomatic number need more test cases than programs with low cyclomatic number. Just like the Halstead's complexity metrics, cyclomatic complexity is also easy applicable to any programming language, but it can be computed earlier in the life cycle of the program than the Halstead's metrics (the program doesn't have to be completed to be able to calculate its complexity value).

Both presented metrics, as well as the others that can be found in the literature, rely on the complete source code, and not on the 'actually visited source code' based on the known values of the program variables. Because this is an important issue, we decided to propose a new metric. It takes into consideration the fact that sometimes, even with a very complex code, the student may simply calculate the output of the source code, if it doesn't depend on large portions of the code. The metric is presented in the following section.

III. Our Source Code Complexity Metric

In this section we define a new metric that can be used to measure the complexity of a program's source code, written in the C++ programming language. The same metric can be extended for usage with source codes written in any programming language.

In our approach, we assume that all of the branch statements of the C++ language (if, while, do-while and for) and the most commonly used C++ operators (the arithmetic operators: +, –, *, / and %; the relational operators: <, >, <=, >=, != and ==; the logical operators: !, && and ||; and the remaining binary operators, such as the assignment operators =, +=, etc.) are assigned a specific weight value. Each of these weight values represents the effort required (from a human) to perform the corresponding operation or execute the corresponding branch statement manually. If the weights of all the operators and branch statements are known, we define the complexity C of a given C++ source code using the following equation:

$$C = \sum_{i=1}^{n} w_i \cdot e_i \qquad (1)$$

where n is the number of lines in the source code, w_i is the weight assigned to line i, and e_i is the number of executions of line i in a single execution of the source code. The weight assigned to a line is the sum of the weights of all the operators and branch statements present in that line.

This metric should, more precisely, calculate the complexity of the source code from perspective of the student's effort to calculate the output of the code. In the next section, we will present our tool that employs this metric.

IV. Our Tool for Calculation of the Proposed Source Code Complexity Metric

For the purpose of calculating our proposed complexity metric for a given source code discussed in the previous section, and to provide a means of automatic production of similar complexity source codes on the basis of the initial code, we have created an appropriate software tool. The tool represents a Java web application that can be accessed using a web browser. Currently, it works only with programs written in the C++ programming language, and it supports Windows and Linux platforms.

The following technologies and libraries were used in the development process of the tool:

- Maven – a software project management and comprehension tool based on the Project Object Model (POM). It can manage a project's life cycle from a centralized XML file [4];
- Eclipse CDT (C/C++ Development Tooling) API [5];
- FreeMarker – a "template engine"; a generic tool to generate text output (anything from HTML to auto generated source code) based on templates [6];
- Gcov – a tool used in conjunction with GCC to test coverage of programs [7];
- Java Server Faces – Java based framework which implements Model View Controller (MVC) design pattern in a stateful manner. The usage of this

framework allowed well refining of the application layers as well as tracking of beans state [8];

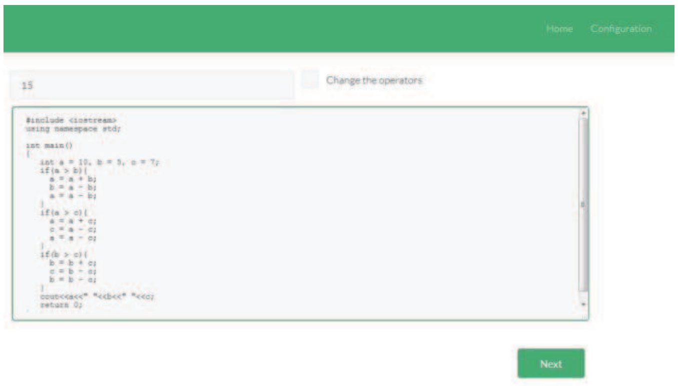
- Spring – used for inversion of control via dependency injection and bean life cycle manipulation [9].

Our tool represents an extension of a tool that uses an initial code to generate a user-specified number of codes by altering literal values and/or operators in it. With the help of the new tool, the newly produced codes can have complexities that do not exceed the complexity of the initial code plus a threshold value. We will refer to the new codes generated by the tool in this way from an initial code as code variations of the initial code.

The combined tool consists of two main parts: 1) Uploading and editing of an initial source code, and generation of code variations; 2) Configuration of the weights associated to each of the operators/statements.

In the first part of the tool, accessed via its home page, there is a wizard that guides the user through the process of generation of code variations. A view of the first step of this process is shown in Fig. 1. In this step, the user can input the initial code – the source code of the program for which he/she wants to generate variations. As can be seen from Fig. 1, in the upper left corner there is a text box in which the user can enter the number of code variations to be generated, and in the upper right corner there is a check box which enables the user to select whether he/she wants modification of the operators in the different code variations. If this check box is not selected, the operators will remain unchanged in all of the generated code variations and will be exactly the same as those in the initial code.

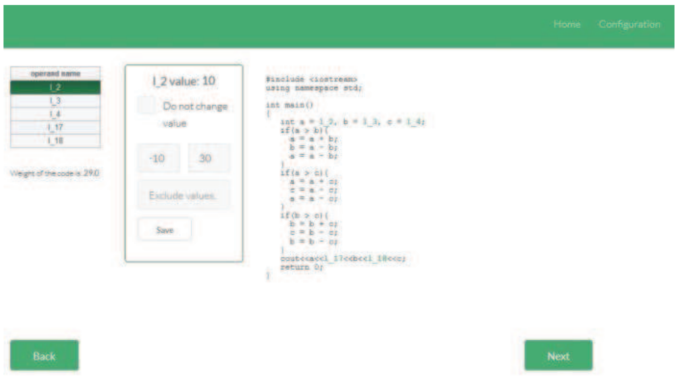
To proceed to the second step of the process, the user has to click on the button labeled 'Next' in the bottom right corner of the page. The wizard will allow this only if no errors have been made in the first step, and otherwise it will show an appropriate error message. Possible error messages at this point are: 'missing value', 'improper value' or 'compiler error'. Codes that cannot be compiled and executed are rejected.



```
#include <iostream>
using namespace std;

int main()
{
  int a = 10, b = 5, c = 7;
  if(a > b){
    a = a + b;
    b = a - b;
    a = a - b;
  }
  if(a > c){
    a = a + c;
    c = a - c;
    a = a - c;
  }
  if(b > c){
    b = b + c;
    c = b - c;
    b = b - c;
  }
  cout<<a<<" "<<b<<" "<<c;
  return 0;
```

Fig. 1. A view of the first step of the process of generation of code variations.

The second step of the process of generation of code variations enables configuration of the domain for the values of the locations of interest in the code (Fig. 2). Locations of interest in a given code are the positions in the code where literal constants (numerical or non-numerical) are present. For numerical locations of interest, the user can configure the range of values from which each of the locations can be filled. Furthermore, the user can explicitly specify a set of values that should be excluded from this range, i.e. which must not appear in the particular location of interest in any of the generated code variations. If the values entered are floating point numbers, then the generated values will contain at most a single digit after the decimal point. For non-numerical locations of interest, the user can configure the character set from which the locations will be filled. The allowed character set can be configured to include (or exclude) digits, lowercase letters, uppercase letters and special characters, by selecting (or leaving unselected) the corresponding check box. Both numerical and non-numerical locations of interest can be left unchanged (by selecting the check box 'Do not change value'), which means that they will keep their default values – the values present at the same locations in the initial code.



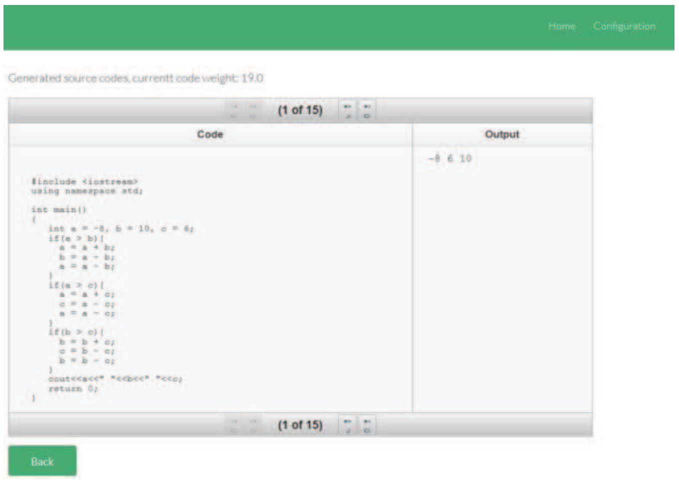
```
#include <iostream>
using namespace std;

int main()
{
  int a = l_2, b = l_3, c = l_4;
  if(a > b){
    a = a + b;
    b = a - b;
    a = a - b;
  }
  if(a > c){
    a = a + c;
    c = a - c;
    a = a - c;
  }
  if(b > c){
    b = b + c;
    c = b - c;
    b = b - c;
  }
  cout<<a<<l_17<<b<<l_18<<c;
  return 0;
}
```

Fig. 2. A view of the second step of the process of generation of code variations.

As shown in Fig. 2, at this moment the user can see the calculated complexity (weight) for the initial code under consideration. The calculated complexity value represents a referent value for the process of generation of code variations. This means that all the code variations that will be generated at the end of the process will have a complexity not greater than the sum of the initial code's complexity value and a predefined threshold. Generated codes with complexities greater than this sum are discarded.

Clicking the 'Next' button in the second step of the generation process starts the actual generation of the code variations. The time required to complete the generation depends on the number of codes that will be generated. After the completion of the codes generation, the wizard brings the user to the third and final step, where he/she can see the results (Fig. 3). The page shows the number of generated codes, and presents the first of them, together with its output and the calculated complexity value. The user can then browse through the other generated codes (by clicking on the left/right arrows above the area where the codes are shown) to see their respective outputs and complexity values.

The second part of the tool provides an interface for configuring the values of the weights that are used in the process of calculation of the complexity of a given code. It can be accessed by clicking on the Configuration hyperlink from the green menu in the upper right corner at any step of the generation process. The interface provides a tabular view of all these weights (Fig. 4), where each row corresponds to a single weight assigned to a particular operator or statement. Each weight has a default value, as can be seen from Fig. 4. The weight values can be easily changed by entering new values in the appropriate fields in the second column of the table. Here, the user can also modify the threshold value by supplying an appropriate value in the last row of the table. In this way, he/she can control the allowed deviation of the complexities of the generated code variations from the initial code's complexity.



Generated source codes, currentt code weight: 19.0

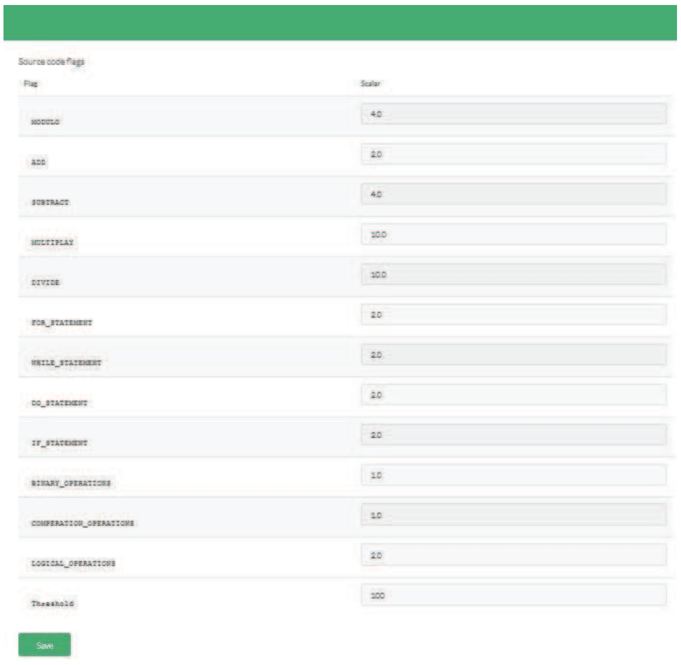
(1 of 15)

Code	Output
(see listing below)	-8 6 10

```
#include <iostream>
using namespace std;

int main()
{
   int a = -8, b = 10, c = 6;
   if(a > b){
     a = a + b;
     b = a - b;
     a = a - b;
   }
   if(a > c){
     a = a + c;
     c = a - c;
     a = a - c;
   }
   if(b > c){
     b = b + c;
     c = b - c;
     b = b - c;
   }
   cout<<a<<" "<<b<<" "<<c;
   return 0;
}
```

(1 of 15)

Fig. 3. A view of the third and final step of the process of generation of code variations.



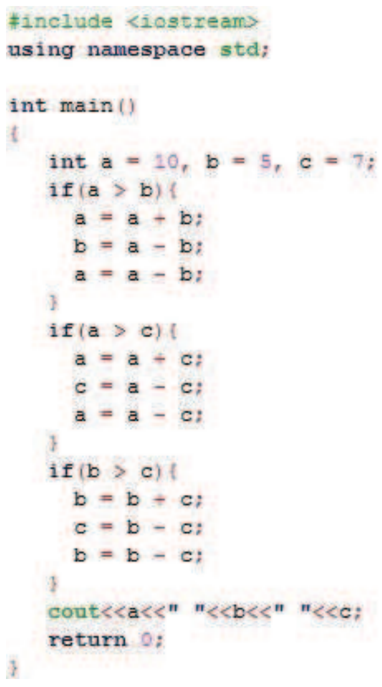
Source code flags

Flag	Scalar
MODULO	4.0
ADD	2.0
SUBTRACT	4.0
MULTIPLAY	10.0
DIVIDE	10.0
FOR_STATEMENT	2.0
WHILE_STATEMENT	2.0
DO_STATEMENT	2.0
IF_STATEMENT	2.0
BINARY_OPERATIONS	1.0
COMPERATION_OPERATIONS	1.0
LOGICAL_OPERATIONS	2.0
Threshold	100

Fig. 4. Configuration of the weight values and the threshold value to be used in the calculation of the complexity of each code.

V. A Case Study

In this section we will describe a case study of an application of our tool on a program's source code in order to calculate its complexity and generate code variations with complexities that have a desired deviation. The example C++ source code is shown in Fig. 5.



```
#include <iostream>
using namespace std;

int main()
{
   int a = 10, b = 5, c = 7;
   if(a > b){
     a = a + b;
     b = a - b;
     a = a - b;
   }
   if(a > c){
     a = a + c;
     c = a - c;
     a = a - c;
   }
   if(b > c){
     b = b + c;
     c = b - c;
     b = b - c;
   }
   cout<<a<<" "<<b<<" "<<c;
   return 0;
}
```

Fig. 5. An example source code for the case study.

Given this initial code, let's suppose that we need to generate 15 different code variations, but without changing any operators in the code. Specifying these input parameters to the tool brings us to the second step of the generation process, as described in the previous section. Here we can specify domains for the locations of interest in the code. We have three numerical and two non-numerical locations of interest in this case, which correspond to the numerical (10, 5 and 7) and the non-numerical literals (the two strings in the 'cout' statement, each of them containing a single white-space character) present in the code. We don't want to modify the non-numerical literals, so we select the corresponding 'Do not change value' check box for each of them. For the numerical literals, we specify the following ranges of integer values as domains:

- [–10, 30] for the value of the location of interest corresponding to the variable *a*,
- [–20, 20] for the value of the location of interest corresponding to the variable *b*, and
- [–30, 10] for the value of the location of interest corresponding to the variable *c*.

We will leave the predefined weight values (as shown in Fig. 4) for the operators and statements unchanged. The calculated complexity of our initial source code with these settings is 29. We will set the threshold value to 20, so that we don’t get code variations with complexities that exceed the value 49.

The results obtained in the final step of the generation process are shown in Table I. As we can see, the desired number of 15 codes have been generated which means that there hasn’t been a rejection of a generated code (due to a too large complexity). The minimum complexity value of a generated code is 9, and the maximum complexity is 29, so only codes with complexities that are less or equal to the initial code’s complexity have been generated.

TABLE I. STATISTICS FOR THE GENERATED CODE VARIATIONS

Observed statistic	Value
Minimum complexity value	9
Maximum complexity value	29
Average complexity value	24.33
Average deviation from initial code’s complexity	6.22
Number of generated codes	15
Number of codes with same complexity as the initial code	10
Number of codes with larger complexity than the initial code	0
Number of codes with smaller complexity than the initial code	5

The results show that even when the initial code is fairly simple (as the one observed in this case study) and the threshold value is set to be relatively small, we may obtain code variations with complexities that may significantly differ from the initial code’s complexity. This proves that it is essential to have a tool for checking the complexities of the generated code variations.

VI. CONCLUSION AND FUTURE WORK

In this paper we described the need for maintenance of consistency of question complexity that appears in the assessment process in the domain of teaching programming, which is especially difficult when working with large course classes. Teachers have to produce questions with same or similar complexity for the students’ tests, in order to achieve objective and fair assessment of the students’ knowledge. A possible and already widely employed solution to this problem is to use automatic production of questions. However, in order to achieve complexity consistency in the process of automatic production of questions that contain source code fragments, which are commonly used in programming course exams, we must have a way to automatically measure the complexity (‘weight’) of source codes.

Further in the paper we described the software metrics that are most commonly used to measure the complexity of a given program’s source code. We explained their advantages and drawbacks, with emphasis on the reason why they’re not suitable to be used for the problem under consideration. Next, we proposed a new metric that considers the source code complexity from a perspective of the student’s effort required to manually calculate the output of the program (if the input is known), and thus, is well suited for the problem. The metric measures the complexity using user-specified weight values assigned to each of the operators and branch statements in the code. We also described our new tool that employs this metric to calculate the complexity of an initial source code, and generate a desired number of new source codes (code variations) with same or close enough complexity (using a user-defined threshold value to control the complexity deviation).

For our future work, we plan to conduct an extensive research in order to determine weight values that will accurately represent the students’ effort required to manually perform the operations and execute the statements in a given source code.

ACKNOWLEDGMENT

The research presented in this paper is partly supported by the Faculty of Computer Science and Engineering, at Ss. Cyril and Methodius University in Skopje.

REFERENCES

[1] B.S. Bloom, M.D. Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl, Taxonomy of educational objectives: The classification of educational goals, Handbook I: Cognitive domain. New York: David McKay Company, 1956.

[2] M.H. Halstead, Elements of Software Science. New York: Elsevier North-Holland, 1977.

[3] T.J. McCabe, “A complexity measure”, IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320, December 1976.

[4] Apache Maven [Online]. Available: https://maven.apache.org/

[5] D. Piatov, A. Janes, A. Sillitti, and G. Succi, “Using the Eclipse C/C++ Development Tooling as a robust, fully functional, actively maintained, open source C++ parser,” IFIP Advances in Information and Communication Technology, pp. 399, 2012.

[6] FreeMarker [Online]. Available: http://freemarker.org/index.html

[7] “Introduction to Gcov,” GCC online documentation [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov

[8] E. Burns et al., “JSR 344: JavaServerTM Faces 2.2”, Java Community Process [Online]. Available: https://www.jcp.org/en/jsr/detail?id=344

[9] Spring framework [Online]. Available: http://projects.spring.io/spring-framework