Increasing the similarity of programming code structures to accelerate the marking process in a new semi-automated assessment approach

Increasing the Similarity of Programming Code Structures to Accelerate the Marking Process in a New Semi-Automated Assessment Approach

Selim Buyrukoglu, Firat Batmaz, Russell Lock
Computer Science
Loughborough University
Loughborough, UK
{s.buyrukoglu, f.batmaz, r.lock}@lboro.ac.uk

***Abstract*— The increased number of students (in higher education) learning programming languages makes the efficient and effective assessment of student work more important. Thus, academic researchers have focused on the automation of programming assignment marking. However, the fully automated approach to marking has its issues. This study provides an approach geared towards the reduction of marking times while providing comprehensive, effective and consistent feedback on novice programmers' code script. To assess novices' code script, a new semi-automated assessment approach has been developed. This paper focuses on the semi-automatic assessment of programming code segments, partially explaining the increasing similarity between code segments using generic rules. The code segments referred to are 'for' and 'while' loops and sequence parts of code script. The initial results and findings for the proposed approach are positive and point to the need for further research in this area.**

***Index Terms*—Automated Assessment, Programming Language, Marking, Feedback, Formative Assessment.**

I. INTRODUCTION

Assessment is essential in teaching and learning, especially for novice programmers. Formative assessment is also important for novices in programming languages or modules. Many programming modules include lab sessions. Programming practice in lab sessions requires good feedback. In recent years, the number of students learning programming has increased in higher education and also secondary school [1]. Thus, assessment is extremely important for providing not only marks, but also efficient and comprehensive feedback. Marking and providing formative feedback increase students' performance, because they are essential parts of the assessment process [2]. However, providing quality feedback is challenging due to the time taken up by existing, largely manual assessment processes.

Manual programming assessment processes are time-consuming and inefficient for assessors. They also tend to increase linearly based on the number of students in the class. Students can get inconsistent and insufficient feedback from manual assessment [3]. Therefore, many assessors have been tempted into using more automated forms of assessment, the primary reason being that a program can be marked and assessed more efficiently by a computer [4]. On the other hand, fully automated assessment systems may return limited feedback, and are heavily dependent on lecturing staff correctly configuring them with model solutions, although they can then provide near instant feedback [5]. Semi-automated assessment systems can be used to address these challenges of manual and fully automated assessment. The term 'semi-automated' is used to mean the cooperation between a human assessor and a computer during assessment [6]. Semi-automated assessment can combine the benefits of Computer-Aided Assessment and lab (tutorial) feedback [6]. Students can get automatic feedback through Computer-Aided Assessment systems based on formative assessment. However, many researchers agree that manual assessment is still necessary to provide effective feedback, because Computer-Aided Assessment systems may miss important parts of code scripts while providing feedback [7][8][9]. Thus, this research focuses on semi-automated assessment.

Novice programmers need a lot of practice programming in order to develop their programming skills. Thus, mainly short answer questions are formatively assessed in this research. In this case, examiners may leave effective comments for each different code segment. This research concentrates on a semi-automated assessment approach for providing feedback comments and marking novice students' programming assignments. The research initially focused only on 'if structures' in code scripts. This paper deals with the assessment of 'sequence' and 'loop (for and while)' parts of code script. In other words, this research focuses on code structures while providing feedback on student code script. Furthermore, students can get good feedback thanks to formative assessment during the tutorial exercises without significant increase to the workload of the examiner. Through increases in formative assessment, novices can more easily develop their programming skills. If humans are more involved, potentially they will be able to give better feedback than a computer; students will able to get detailed feedback from humans.

The rest of this paper is structured as follows: the next section introduces related works in the field. This is followed by Section III, presenting the proposed semi-automated assessment approach. Section IV discusses the details of codifying rules. Section V is about the data collection. Section

VI includes a real-world case study evaluation of the approach developed and discusses the results. Section VII provides conclusions and outlines the potential for future work in this area.

II. Related Work

Pears et al. [10] classify the tools that support teaching programming into four groups: (1) visualisation tools, (2) Automatic Assessment tools, (3) programming support tools, and (4) microworlds.

Automatic programming assessment has a long history. Computer scientists in education have been interested in it since the 1960s and it is still an active research field. Its main goals are to implement an automatic assessment tool to provide consistent feedback and to alleviate examiners' workloads. Many automatic assessment tools and systems have been developed so far. Measurement values are one of the basic requirements for the automated assessment of programming assignments [11].

Automatic assessment tools are generally developed to check the correctness of the program running, e.g. Course-Marker [12], Boss [13]. Correctness is checked through a comparison of the student's program solution and a model solution. Also, some automatic assessment systems aim to compare internal data representations [14] or try to develop by testing parts of automatic assessment tools to provide more detailed feedback [15]. Furthermore, these tools can also analyse different criteria of program code, including coding style, program efficiency and complexity.

These tools can be used for different purposes. They can be used not only in support of a grading process, but also to generate an initial evaluation indicating problems in students' code. Furthermore, more than one examiner can use the same tool to provide feedback; it will summarise the examiners' feedback to check for consistency. If there is any inconsistency between different people's feedback, some types of tool allow examiners to modify feedback [16].

Our paper proposes a semi-automatic assessment approach which utilises automatic and manual methods. We believe that our approach opens a new line of research in Automatic Assessment. Many semi-automated assessment approaches have been developed so far, such as [17][18][7][19][20]. These approaches aim at increasing the human role in assessment systems in order to provide detailed feedback. In the testing phase, it gives the examiner the opportunity to view the students' source code [18] and comment upon it. Other systems run the code scripts [17][7][19][20]. If the code scripts do not work, the human becomes involved. If they do work, the computer automatically provides feedback. However, this feedback can be lacking in detail, because they generally focus on running of the code rather than providing detail feedback and also, if the code script does not work, the examiner fixes the errors to run the code. On the other hand, in the proposed approach, the examiner does not fix the students' code scripts and can be involved throughout the assessment process until the students get feedback. Systems that provide feedback are generally based on the correction of the code segments. In this process, model answers are used to measure the correctness of students' code segments. On the other hand, if a student's program does not compile and gives syntax errors, the systems allow lecturers to fix these errors and provide feedback.

These semi-automated assessment systems can provide useful feedback for students; however, examiners' workloads cannot be alleviated using these systems, because they must view each code individually and make comments to develop the feedback. In this case, if the systems saved lecturers' comments, they could have been designed to use them again; lecturers' workloads could have been reduced. In addition to this, systems generally generate feedback based on the correction of code and syntax errors, rather than logical errors and code structures. Therefore, this research focuses on providing feedback on code structures and logical errors rather than basic feedback. Furthermore, this study aims at a significant decrease in examiner's workload through the following proposed semi-automated assessment approach.

III. The Semi-Automatic Assessment Approach

There is a variety of research focusing on fully automated programming code assessment which provides instant feedback and reduces the workload of the examiner [21]. However, few studies focus on semi-automated assessment systems. The aim of this research is to provide better feedback compared to fully automated assessment systems. Semi-automated assessment can provide more effective feedback, through a human assessor, than fully automated assessment systems if the approach has been developed systematically. Fig. 1 shows the proposed assessment approach cycle.

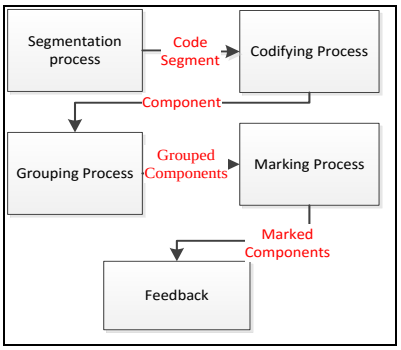

Fig. 1. Proposed assessment approach cycle

A. Segmentation Process

This process parses the code script in terms of code structure to derive code segments. The study consists of two type of code segments, including 'loops' and 'sequences'. The most important part of this process is parsing, which needs to be applied based on code structures. For example, the following code script, written in python, includes two code segments:

```
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:
  print 'Current fruit :', fruit
```

The first code segment refers to a 'sequence type', which means that the code segment does not include any control

structures, functions etc. In this example, the ‘sequence type’ refers only to the first line. The second code segment starts with a ‘for loop’ line and finishes with the last line of the loop.

B. Codifying Process

Sometimes, code segments from different code scripts can be similar or even identical, though the longer the segment, the less likely it is to be identical. The aim of the codifying process is to normalise code segments based on code structure. Code segments are called components after the codifying process, which is described in Fig. 1. In this process, generic rules can be applied to code segments. Thus, similar code segments can be identified and put into the same group. The generic rules will be discussed in detail in Section IV. The number of rules can be also extended according to programming language requirements. However, these rules should not cause any errors, such as syntax or semantic errors; that is, the running of the code segments should not change due to the generic rules.

C. Grouping Process

The components can be grouped by the system. String matching is the main part of the grouping process. The result of the string match directly affects group numbers. Identical components can be put into the same group.

A group can include more than one component. A component may be a ‘for’ loop, a ‘while’ loop, a ‘do-while’ loop, a ‘sequence part of code script’ etc. for the purposes of this research. Code script generally includes more than one component. For example, if a code script consists of ‘sequence type’ and ‘for loop’ components together, they can be put into separate groups, because they not only refer to different code structures, but also are not identical. In the following example, two different groups are created by the grouping process. Fig. 2 illustrates two groups of components (component A and component B).

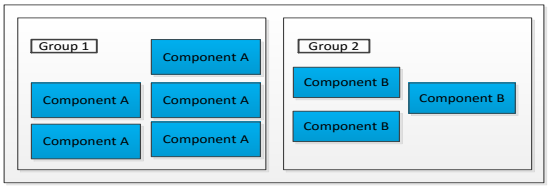

Fig. 2. Group 1 and group 2 containing different components

Group 1 and Group 2 have five components A and three components B respectively. Letters were used in Fig. 2 instead of real program code. Components can be marked after this grouping process.

D. Marking Process

The grouped components from the previous section are marked in this process. The examiner needs to mark only one member from each group, rather than all of the members, and the rest of the members from each group can be automatically marked by the system using the examiner’s comments.

In this process, the examiner may also check all marked components to ensure the correctness of the comments. Therefore, components should be presented clearly to the examiner. The order of identical components in different code scripts can vary, in which case students could get incorrect feedback. Thus, in the marking process, whole original code scripts, including all components, can be displayed to examiners through an assessment tool. They may check the comments on each component individually. Moreover, they can also make additional comments if necessary. Before the examiner checks and accepts the comments on a component, it can be highlighted with any colour by the system, which shows that the correctness of the marked and commented upon component has not yet been accepted by the examiner. At the same time, all identical components from other code scripts are also highlighted using the same colour code. After the examiner accepts the comments on the component, it and identical components from other code scripts are highlighted in a different colour by the system, showing that they have received the same comments and these comments have been accepted by the examiner. Meanwhile, identical components can be highlighted in yet another colour if their order is different from the order accepted by the examiner. In this case, the examiner can check the components again to provide correct feedback. Then, each component can be highlighted in the same colour and can be marked as assessed. In this study, the order of components is very important in terms of the correctness of the feedback.

Marked and commented upon components can be stored in a database. The ideal solution for previously marked components can be used to mark similar components in different students’ code scripts. Therefore, this can decrease the examiner’s assessment workload and increase automation.

IV. Normalisation of Code Segments Using Codifying Rules

In this section, the codifying rules for increasing similarity between code segment structures are explained. This paper focuses on two different code segment structures, including ‘loops’ and ‘sequence types’. The loops can be ‘for’, ‘while’ or ‘do-while’ loops. Parts of code segments may be changed by the application of the rules. However, the rules do not affect the running of the program. The rules have been created based on the python programming language, but they can be added to if necessary. In this paper, all the used arguments, such as variables, operators and the values of variables, are considered identical among different students’ code scripts.

A. Rule I

Rule I is applicable for ‘sequence type’ code segment structures and block parts of loops if they refer to sequence type. The order of lines without print messages and equation lines in a code segment can be alphabetically arranged to increase the similarity between code segments. If a code segment has already been written in alphabetical order, the rule cannot be applied. Otherwise, it is applicable. The following example shows the usage of Rule I.

TABLE I. EXPLANATION OF RULE I

Non-Codified			Codified	
Segment 1	Segment 2		Component 1	Component 2
a = 3 **b = 4** print a + b	**b = 4** **a = 3** print a + b		**a = 3** **b = 4** print a + b	**a = 4** **b = 3** print a + b

The order of lines in Segment 2 is not alphabetical in Table I. The order can be swapped so the lines are in alphabetical order in the Component 2 column of Table I.

B. Rule II

Rule II is applicable to 'sequence type' code segment structures and block parts of loops if they refer to sequence type. If any code segment line has an equation, the argument order in the equation can be arranged according alphabetically. Thus, similarity among code segments can be increased. Table II shows an example of Rule II.

TABLE II. EXPLANATION OF RULE II

Non-Codified			Codified	
Segment 1	Segment 2		Component 1	Component 2
a = x + y print a	**a = y + x** print a		**a = x + y** print a	**a = x + y** print a

The equation in Segment 2 is not in alphabetical order. Its order can be rearranged based on Rule II. Then, Segment 2 can be referred to as Component 2, which is in alphabetical order.

C. Rule III

Rule III is applicable only to the condition parts of 'while' and 'do-while' loop code segment structures. Condition parts of code segments generally include arguments in a specific order. For example, the variable may be written first, and then the math operators and the value of the variable will be written after. Rule III can be used to swap the order of arguments in a condition if it is different from the commonly used order. Thus, similarity can be increased using Rule III. The following Table III displays an example of Rule III.

TABLE III. EXPLANATION OF RULE III

Non-Codified			Codified	
Segment 1	*Segment 2*		*Component 1*	*Component 2*
(x < 3)	***(3 > x)***		***(x < 3)***	***(x < 3)***

In Table III, the argument order in Segment 2 is different to the most commonly used order. The order can be swapped so it becomes Component 2 through Rule III.

D. Rule IV

This rule is applicable to the condition parts of loops if they have more than one condition. The condition parts of code segment structures are generally written according to certain rules, such as the first condition being written first, and then logical operators and other conditions being written later. In this case, the order of conditions can be made alphabetical if it is not already in alphabetical order. Thus, the conditions can be given standard form using Rule IV. If they have already been ordered alphabetically, Rule IV will not be applicable. The following example shows the usage of Rule IV.

TABLE IV. EXPLANATION OF RULE IV

Non-Codified	
Segment 1	Segment 2
(x < 3) and (y <4)	**(y < 4) and (x < 3)**
Codified	
Component 1	Component 2
(x < 3) and (y <4)	**(x < 3) and (y <4)**

The condition order in Segment 2 is wrong according to Rule IV. The order of the conditions can be swapped through the rule. After applying Rule IV to Segment 2, it can be referred to as Component 2, as shown in Table IV; that is, its order can be changed alphabetical.

E. Rule V

Rule V is applicable to any code segment structure. If there are whitespace in code segment, it is disregarded for the purposes of similarity measurement. Table V shows an example on Rule V.

TABLE V. EXPLANATION OF RULE V

Non-Codified			Codified	
Segment 1	Segment 2		Component 1	Component 2
a = x + y print a	a = y + x *whitespace* print a		a = x + y print a	a = x + y print a

The whitespace in Segment 2 can be ignored based on Rule V and the structure similarity between Segment 1 and Segment 2 subsequently increases. In other words, Segment 2 can be referred to as Component 2 in Table V.

F. Rule VI

This rule is applicable to any code segment structure. If there are any comment line(s) or comment(s) in code segment, they can be ignored by the system to increase the similarity between code segment structures. Table VI displays an example on Rule VI.

TABLE VI. EXPLANATION OF RULE VI

Non-Codified			Codified	
Segment 1	Segment 2		Component 1	Component 2
a = x + y print a	a = y + x *#comment* *# comment line* print a		a = x + y print a	a = x + y print a

In Table VI, the comment line and comment in Segment 2 can be ignored by the system. After applying Rule VI to Segment 2, it can be referred to as Component 2, as shown in Table VI.

G. Rule VII

This rule is applicable to the condition part of loops. If any condition is not written in parenthesis, the system can assume that the structure of condition is equivalent to any structure of condition in parenthesis. Table VII displays an example on Rule VII.

TABLE VII. EXPLANATION OF RULE VII

Non-Codified			**Codified**	
Segment 1	Segment 2		Component 1	Component 2
while **(**a <11**)**: print a	while a < 11: print a		while **(**a < 11**)**: print a	while **(**a < 11**)**: print a

In Table VII, the structure of condition in Segment 2 can assume equivalent to the structure of condition in Component 2 based on Rule VII. Then, Segment 2 can be referred to as Component 2.

In the codifying process, some part of components can be changed to increase the similarity between code segments through the codifying rules such as ignoring the whitespace lines etc. However, original code script can be represented to examiner in the marking process. In this case, examiner's comments can be more realistic.

Group numbers can be reduced through these codifying rules, which also helps in decreasing the workload of the examiner. The codifying rules can be added to base on the programming language requirements. Well prepared questions can also limit students' answers. Thus, similar code segments can easily be captured from different code scripts.

V. DATA COLLECTION

Two different questions were asked of students taking semester one (2014) of the introduction to programming module at Loughborough University. The lab exam, which 55 students attempted, asked about the usage of 'sequence type' and 'loop (for-while)' structures. Questions should ask about specific parts of programming structures. The questions were designed to ask about and assess certain programming structures. The questions asked each assessed one or two programming structures. Question 1 checked students' understanding of the usage of sequence part code scripts, while Question 2 checked their usage of 'loop' structures. Questions can be designed to check all the different structures that exist. Furthermore, question scenarios are very important in formative assessment, which is part of the learning process. In this case, the students used the python programming language to complete the tasks.

VI. ANALYSIS AND DISCUSSION

In this approach, code scripts are automatically parsed and then manually codified and grouped. Initially, the code scripts were parsed, which refers to the segmentation process based on code structure. Then, each of them was manually codified during the codifying process. In this process, the similarities between code segments are increased based on the proposed generic rules in Section IV. For example, the orders of statements in code segments were fixed manually, using Rule 1. Then they were considered as components rather than code segments. These components were later manually grouped before the marking process. This process was applied to both Question 1 and Question 2. Fig.3 shows the group numbers and populations for Question 1.

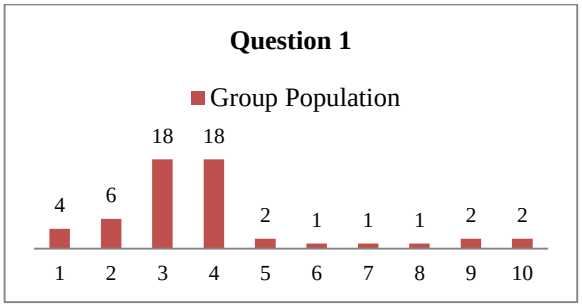

Fig. 3. Group information for question 1

According to Fig. 3, 10 groups were created from 55 components. The 55 components were derived after parsing the 55 students' code scripts. In other words, 10 of 55 components need to be marked by the examiner, which equates to 19% of the components. The rest of the components, 45 of the 55, which equates to 81% of components, can be assessed by the proposed assessment system. For example, group 1 has four components: one of the four components needs to be assessed by the examiner, while the remaining three components can be assessed by the proposed assessment system.

Code scripts for Question 2 were also analysed. Fig. 4 shows the group numbers and populations for Question 2.

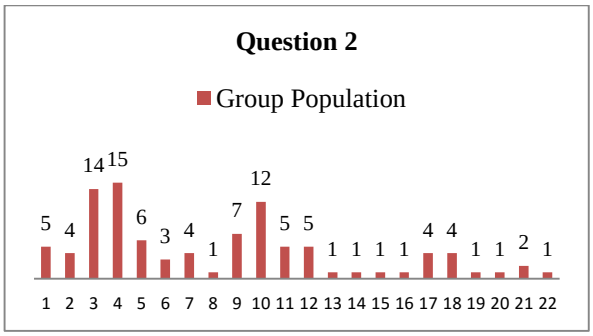

Fig. 4. Group information for question 2

As can be seen from Fig. 4, 22 groups were created from 98 components. The 98 components were derived by parsing the 55 students' code scripts. In other words, 22 of 98 components need to be marked by the examiner, which equates to 22% of components. The rest of the components, 76 of the 98, or 78% of components, can be assessed by the proposed assessment system.

At the end of the assessment process, two questions can be assessed through the proposed assessment approach. Only 32 components need to be assessed by the human marker; the rest of the 121 components can be assessed by the proposed assessment system. These highlighted numbers are related to the example introduced in this paper.

According to the results, the proposed assessment system can save significant amounts of the human assessor's time compared to manual assessment. The results show that up to 75% of components can be assessed by the proposed system. This is an advantage for the human assessor. However, this study has some limitations, such as having only analysed novice students' code scripts in short question form. Not only are novices' code scripts basic, but they are also limited by the examiner asking well prepared, limited questions. That is, similar code segments are generally obtained.

Many automatic assessment systems have been developed to provide instant and consistent feedback on students' code scripts [22]. Students can fix their errors based on the provided feedback. However, semi-automated assessment systems can provide not only consistent feedback, but also more effective feedback than fully automated assessment systems, if they are developed systematically, like the assessment approach proposed in this paper. Moreover, generally, human collaboration with machines can positively affect feedback quality.

VII. Conclusion

This study has introduced a semi-automated assessment approach. This approach was applied manually. It has four important part: segmentation, codifying, grouping and marking. Code scripts were parsed based on code structures and then codified according to codifying rules to increase the commonality between code segments. In the grouping process, components were grouped together. The codifying rules, designed to increase the similarities between code segments, have been highlighted and discussed. Preparing the questions is extremely important for obtaining similar code segments in different code scripts, as briefly discussed in this paper. It may also help improve the consistency and reliability of the assessment system. It may potentially decrease the marking time. The proposed approach is feasible according to results of the case study, because the human assessor only needed to assess 32 out of 153 components for two questions. In other words, 27% of components were assessed by the human assessor. To conclude, the proposed approach can be effective if it is used for formative assessment, including lab exams, although it does not provide instant feedback.

In the present study, the code script was analysed manually. A semi-automated assessment tool could be implemented based on the proposed approach. In addition to this, the proposed semi-automated assessment approach could also be used in class tests for summative assessment.

References

[1] M. Resnick et al., "Scratch: programming for all", Communications of the ACM, 2009, 52(11): 60-67.

[2] S. Davies, "Effective Assessment in a Digital Age A guide to technology-enhanced assessment and feedback", 2010.

[3] J. Orrell, A. Havnes, L. McDowell, "Assessment beyond belief: the cognitive process of grading", Balancing dilemmas in assessment and learning in contemporary education, 2008, pp.251-63.

[4] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments", Computer science education, 2005, 15(2): 83-102.

[5] B. Cheang, A. Kurnia, A. Lim, W.C. Oon, "On automated grading of programming assignments in an academic institution", *Computers & Education*, 2003,41(2): 121-131.

[6] D. Herding, U. Schroeder, "Using capture & replay for semi-automatic assessment", CAA 2011 International Conference, 2011.

[7] D. Insa, J. Silva, "Semi-Automatic Assessment of Unrestrained Java Code", Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ACM, 2015: 39-44.

[8] G. Sindre, A. Vegendla, "E-exams and exam process improvement", Norsk Informatikkonferanse (NIK), 2015.

[9] M. Pozenel, L. Furst, V. Mahnic, "Introduction of the automated assessment of homework assignments in a university-level programming course", Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on. IEEE, 2015: 761-766.

[10] A. Pears, S. Seidman, C. Eney, P. Kinnunen, L. Malmi, "Constructing a core literature for computing education research", ACM SIGCSE Bulletin, 2005, 37(4): 152-161.

[11] S. Gupta, S.K. Dubey, "Automatic assessment of programming assignment", Computer Science & Engineering, 2012, 2(1):67.

[12] C. A. Higgins, G. Gray, P. Symeonidis, A. Tsintsifas, "Automated assessment and experiences of teaching programming", Journal on Educational Resources in Computing (JERIC), 2005, 5(3): 5.

[13] M. Joy, N. Griffiths, R. Boyatt, "The BOSS online submission and assessment system", ACM Journal of Educational Resources in Computing, 2005, 5(3):2.

[14] R. Saikkonen, L. Malmi, A. Korhonen, "Fully automatic assessment of programming exercises", ACM Sigcse Bulletin, 2001, Vol. 33, No. 3, pp. 133-136, ACM.

[15] S.H. Edwards, "Improving student performance by evaluating how well students test their own programs", Journal on Educational Resources in Computing (JERIC), 2003, 3(3), p.1.

[16] T. Ahoniemi, T. Reinikainen, "ALOHA-a grading tool for semi-automatic assessment of mass programming courses", Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006, 2006, pp. 139-140, ACM.

[17] K. Ala-Mutka, H. Järvinen, "Assessment Process for Programming Assignments", Null. IEEE, 2004: 181-185.

[18] D. Jackson, "A semi-automated approach to online assessment", ACM SIGCSE Bulletin ACM, 2000, 32(3): 164-167.

[19] S. H, Tung, T. T. Lin, Y. H. Lin, "An Exercise Management System for Teaching Programming", Journal of Software, 2013, 8(7): 1718-1725.

[20] N. Yusof, N. A. M. Zin, N.S. Adnan, "Java programming assessment tool for assignment module in moodle e-learning system", Procedia-Social and Behavioral Sciences, 2012, 56: 767-773.

[21] G. Conole, B. Warburton, "A review of computer-assisted assessment", *Research in learning technology*, 2005, *13*(1).

[22] J. C. Caiza, A. Ramiro, "Programming assignments automatic grading: review of tools and implementations", 2013.