Semi-Automatic Assessment Approach to Programming Code for Novice Students

Selim Buyrukoglu, Firat Batmaz and Russell Lock
Department of Computer Science, Loughborough University, Epinal Way, Loughborough, U.K.

Keywords: Automatic Assessment, Programming Language, Intelligent Tutoring System, Online Assessment.

Abstract: Programming languages have been an integral element of the taught skills of many technical subjects in Higher Education for the last half century. Moreover, secondary school students have also recently started learning programming languages. This increase in the number of students learning programming languages makes the efficient and effective assessment of student work more important. This research focuses on one key approach to assessment using technology: the semi-automated marking of novice students' program code. The open-ended, flexible nature of programming ensures that no two significant pieces of code are likely to be the same. However, it has been observed that there are a number of common code fragments within these dissimilar solutions. This observation forms the basis of our proposed approach. The initial research focuses on the 'if' structure to evaluate the theory behind the approach taken, which is appropriate given its commonality across programming languages. The paper also discusses the results of real world analysis of novice students' programming code on 'if' structures. The paper concludes that the approach taken could form a more effective and efficient method for the assessment of student coding assignments.

1 INTRODUCTION

Automation in the assessment of programming exercises has become an important and also increasingly complex consideration in the marking of students' programming exercises (Rubio-Sánchez, 2014). Many students learn programming languages over a number of years of study. In recent years, younger people have also started to learn programming languages in Further Education (FE) and secondary schools (Resnick et al., 2009) rather than just in Higher Education (HE). Instead of using general high-level languages such as Java or C++ to teach younger, novice students, specialist languages have been developed, with one of the more popular being scratch (Meerbaum-Salant et al., 2013).

The manual programming assessment process is not efficient for assessors, partly because it scales linearly and suffers significant duplication of effort given the commonality of many fragments of code (Palmer et al., 2002). Many researchers therefore focus on automatic assessment, because each program could be assessed and analysed more efficiently by computer if a computer could be configured to do so (Ala-Mutka, 2005). Sharma et al. (2014) indicated that students' programming code can be analysed dynamically or statically. In dynamic analysis, each student's program code is executed and then the result is checked to ascertain the correctness of the program. By contrast, during a static analysis, the code is examined and evaluated without running the program.

The purposes of automatic assessment systems are different, and can be summative or formative. Scriven, (1967) stated that at the end of the learning period, students' learning is measured and their own achievement, which is reported through a summative assessment. Melmer et al. (2008) specified that formative assessment is directly related to the enhancement of a student's education based on feedback. Students may understand their learning more deeply through formative assessment (Clark, 2011). Assessment systems can assess syntax errors, logic errors and semantic errors relating to the source code. Syntax errors represent incorrect statements in programming languages (Kaczmarczyk et al., 2010). Semantic errors involve programming code which is syntactically correct; buy includes incorrect conditions (Schmidt, 2012). Programs with logic errors run without faults but produce incorrect results (Spohrer and Soloway, 2013).

Buyrukoglu, S., Batmaz, F. and Lock, R.
Semi-Automatic Assessment Approach to Programming Code for Novice Students.
In *Proceedings of the 8th International Conference on Computer Supported Education (CSEDU 2016) - Volume 1*, pages 289-297
ISBN: 978-989-758-179-3

Students' programming assignments were mostly assessed manually before the early 2000s (Cheang et al., 2003). Within manual approaches to assessment, each programming assignment is assessed only by a human (Cheang et al., 2003). Both automatic and manual assessments have some key advantages and disadvantages. With automatic assessment, students can expect to receive feedback in a shorter period of time compared to that of manual assessment (Foxley et al., 2001). However, automatic assessment systems may return limited feedback, and are heavily dependent on lecturing staff correctly configuring systems with model solutions. Manual assessment can be effective for students in terms of detailed feedback if the examiner assesses students' assignments as well; that is, if examiners focus on the strengths and weaknesses of the students' programming assignments, they can provide comprehensive feedback. However, manual assessment is an inefficient process, and can lead to minor inconsistencies in commenting accuracy and depth (Cheang et al., 2003). Jackson, (2000) indicated that the merger of both automatic and manual assessment is a beneficial solution, because students' programming assignments can be assessed not only in a short time, but also with more detailed feedback than if based on assessment by computer alone. Such an approach is termed semi-automatic assessment (Jackson, 2000). This research focuses on semi-automatic assessment in order to give a better quality of feedback whilst retaining much of the efficiency increase associated with automated approaches. The research initially deals with the assessment of 'if' structures in order to prove the theory behind the approach, because novice programmers initially learn 'if' structures.

The structure of the paper is as follows: the next section introduces related works in the field. Section 3 presents our approach, including potential identifying, codifying, grouping and marking processes. Section 4 includes a real-world case study evaluation of the approach developed. Section 5 discusses the current issues and limitations posed by the outlined approach, and the final section provides conclusions and outlines the potential for future work in this area.

2 RELATED WORK

In literature, there have been many approaches proposed for the automatic assessment of programming code. This paper will examine five key existing approaches which have influenced our own.

In the system developed by Wang et al. (2007), semantic similarity is measured between students' code and model answers to provide the grade. The semantic similarity measurement includes program size, structure and statement matching. In this approach, the student's program code structure is standardised using specific rules to measure the semantic similarity. Otherwise, the student program and the model answers cannot be matched in terms of the different ordering of statements or code structure in the assessment process. Thus, some rules are created in the system to standardise the students' program code. The aim of the standardisation is to eliminate unimportant syntactic variations from a feedback perspective, and so reduce the number of model answers, improving the program matching in this system. For example, expressions, control structures, and function invocation are standardized by the system. Then, variables are renamed, redundant statements are removed and statements are reordered to improve automatic matching by the system. Lastly, the semantic structures of student program and model answers are matched, and then the student gets the grade.

The aim of the system developed by Sharma et al. (2014) is to teach students only using an 'else-if' structure, rather than a broader range of control structures. Within the approach, a student's 'else-if' structure, a block of programming that analyses variables and chooses the direction in which to go based on given parameters, is normalised in order to compare it with a model answer. In order to normalise the 'else-if' structure, each condition, nested logic operator, arithmetic clause and relational operator is automatically converted to a clearer form using rules applied by the system. After the normalisation of each statement in the 'else-if' structure, the system checks the ordering of the conditions.

Scheme-Robo is a system developed by Saikkonen et al. (2001). In this system, students can resubmit their own assignments a certain number of times. Students' code structures are checked by the system using an abstract tree for the code writing exercises. This tree includes the general structure of code or special sub-patterns. That is, the system executes the tree to ascertain the correctness of the students' code structure. The system provides feedback on failed parts of code as comments.

Jackson, (2000) indicated that human comment is important to provide comprehensive feedback in the semi-automatic assessment systems, and so he developed a semi-automatic assessment. The system

assesses the students' program source code. In the testing phase, the system gives the lecturer the opportunity to view the students' source code. The system also asks the lecturer questions regarding students' program code as part of the assessment process. The examiner gives an answer to questions posed choosing one possible answer from the listed answers such as awful, poor, fair, very good etc., and then the assessment process continues. It also applies software metrics to students' programs. Lastly, students get feedback on his/her exercises.

In the system developed by Joy et al. (2005), the correctness, style and authenticity of the student program code is assessed. It is designed for programming exercises. Students can submit their programs using the BOSS system (a submission and assessment system) (Joy et al., 2005). In the feedback process, a lecturer tests and marks the students' submissions using BOSS. The system also allows lecturers to get information on students' results according to the automatic test applied and to view original source code. Thus, the examiner can then give further feedback in addition to the system's feedback. At the end of the assessment, the student gets feedback including comments and a score, rather than just a score.

2.1 Discussion of Related Work

In the related work section five studies were introduced in terms of their strengths and weaknesses. Although some of them may provide sufficient feedback if correctly applied, they are not designed to significantly alleviate the workload of the examiner. That is, providing feedback may impact negatively on the time taken by the examiner to assess student work. The workload of the examiner entirely depends on the approach of the assessment systems. In addition to this, the workload of the examiner also depends on the length of the code script which could be short or long.

The systems of Wang et al. (2007) and Sharma et al. (2014) and Saikkonen et al. (2001) focus on student programming code structures, which can be useful, but limiting in terms of feedback. While the systems of Wang et al. (2007) and Saikkonen et al. (2001) focus on the whole code structure in their own systems, the system of Sharma et al. (2014) covers only the ordering of conditions in the 'else-if' structure. The aim of the standardisation of the code structure in the system of Wang et al. (2007) is to reduce the number of model answers. Furthermore, code structure is standardised to provide grade students' code rather than to provide comment (feedback) on the code structure. On the other hand, the system of Saikkonen et al. (2001) assesses the return values instead of actual output strings because of the differences in wording in students' answers. In other words, the system focuses on the execution of abstract tree. In this case, the system of Saikkonen et al. (2001) may not provide comprehensive feedback for students. Also, Sharma et al. (2014) system can be used only for 'else-if' structures, which is effective for providing feedback to novice programmers, but only on 'else-if'. However, the theory behind the system could allow it to handle other control structures, loops and functions in the future. Moreover, the quality of feedback could have been enhanced by inclusion of a human in the assessment process.

The system developed by Jackson (2000) and Joy et al. (2005) highlights the importance of human in the assessment process in providing comprehensive feedback. In Jackson (2000) system the examiner is part of the assessment process; however, the examiner is used after the assessment process in Joy et al. (2005) system. In these systems, humans check each student's code separately. Therefore, the systems cannot reduce the workload of examiners significantly, although they can provide sufficient feedback using these approaches. One significant drawback to Joy et al. (2005) system is that while examiners can give additional comments to students, the system could also potentially provide inconsistent comments (as this is not checked by the system). It is this automatic reuse of feedback provided for given segments of code that would have allowed greater consistency and efficiency to be achieved. On the other hand, in Jackson (2000) system, the examiner chooses one comment from the suggested comments. However, the system cannot provide comprehensive feedback because the examiner cannot add comments to the student's code.

To conclude, the discussed assessment studies intended to provide sufficient feedback and reduce the workload of the examiner. However, they have generally focused on whole code segments rather than control structures, loop, functions etc. Thus, they have generally provided superficial feedback although some of them reduce the workload of the examiner. Moreover, these discussed studies generally based on the semantic similarity. The proposed approach also related to semantic and structure similarity. The main difference between them is that the proposed approach does not need model answer(s) although the discussed studies do. Therefore, the proposed approach parses the whole

code script based on the repetitive parts of code structures such as sequence part of code segments, 'if' control structures, loops and functions etc. before the examiner providing feedback on them. The following section discusses the approach of this research.

3 APPROACH

This section describes the proposed approach, which is based on analysing the source code of novice programmers. It focuses on commenting the repetitive elements of students' program code. All programming languages could potentially be assessed using the proposed approach. That is, this approach can be applied equally well independent of programming language. Previous observations by the paper authors of student code indicate that students' code structures generally contain similar code segments. Table 1 shows examples of code segments.

Table 1: A code example.

Name	Program Code
A	`if (x==5){ print 'x equals to 5'}`
B	`else{ print 'x is not equals to 5'}`

In Table 1, code segments A and B together form code referred to as 'AB'. In this example, code segment A refers to an 'if' structure, including the condition and block parts. The condition part is '(x==5)', while the block part is 'print 'x equals to 5''. Table 2 gives example on same code segments among students code.

Table 2: Programming codes.

Student 1	Student 2	Student 3	Student 4
A	A	A	A
B	B	D	B
C	C	C	C

Each of the letters in Table 2 refers to a common code segment. To simplify the explanation, the code segments are illustrated as letters. This approach has four key processes, which are identifying, codifying, grouping and marking. Figure 1 shows the process model of the proposed semi-automatic assessment approach.

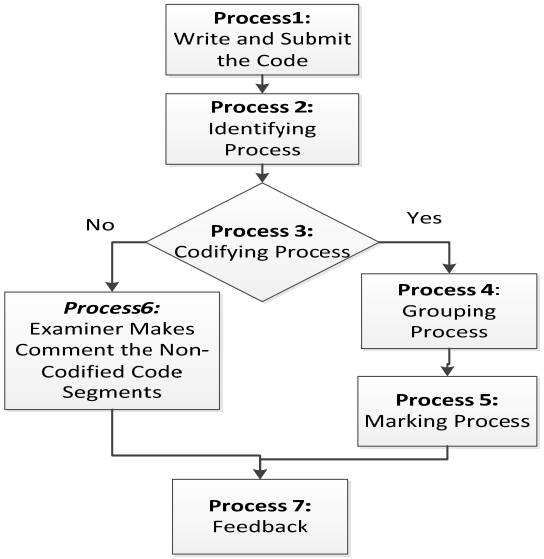

Figure 1: Processes model of the proposed Semi-Automatic assessment approach.

The processes included in Figure 1 outlined further in the bullet points below.

- **Process 1:** Students attempt to code an answer to a question and then submit it.
- **Process 2:** Similar code segments are identified by the system.
- **Process 3:** Similar code segments are automatically codified by the system. For example, in Table 2, four different similar code segments are codified: the components A, B, C and D. (After the codifying process, they are called as components).
- **Process 4:** Components are automatically grouped by the system in this process.
- **Process 5:** The examiner comments on each different component once for each group. Then, the examiner comments are utilised to mark the rest of the component in the same group by the system.
- **Process 6:** If any similar code segments cannot be automatically codified by the system, they can be marked and commented on manually by the examiner.
- **Process 7:** Students' results can be given as feedback.

Using this assessment process, examiners' workloads may be partially alleviated. On the other hand, examiners have to ask very clear questions to find similar code segments in different students' code. Thus, the question type is very critical within this approach. The following sections discuss the

identifying, codifying, grouping and marking processes in detail.

3.1 Identifying Process

Different students' code can contain similar code segments. Their similarities can be identified in this process. This process refers to Process 2 in Figure 1. Similar code segments can also include similar control structures, conditions and block parts. The following bullet points explain the details of similar code segments.

- **Similar Control Structure:** If students' code segments include only the same control structures, and disregard the condition and block parts, they are considered similar control structures.
- **Similar Condition:** If the numbers of arguments in the condition parts of structures are the same, regardless of the control structure name and block parts, they are considered similar conditions.
- **Similar Block:** If the block parts of control structures include the same line, they are considered similar block parts.

In this approach, if different students' code includes similar control structures, similar conditions and similar block parts all together, they are considered similar code segments (i.e. similar(code segment)= similar(control structure) + similar(condition) + similar(block)). In Table 3, two similar code segments are illustrated.

Table 3: Examples of similar code segment.

Name	Similar Code Segment

Similar Code Segment-1	`if ((x = 0)and(t = 0)){`
	`y = x + 1`
	`z = t - 1`
	`print 'y is positive'`
	`print 'z is negative'}`

Similar Code Segment-2	`if ((t = 0)and(x = 0)){`
	`z = t - 1`
	`y = x + 1`
	`print 'z is negative'`
	`print 'y is positive'}`

In Table 3, two code segments are considered similar code segments because they have similar code structures, similar conditions and similar block parts. Both of them include only 'if' structures, regardless of the condition and block parts of them, and so they are considered similar code structure. In the condition part of the two similar code segment entities, it is obvious that the number of argument remain unchanged. Therefore, those two condition parts can be considered as similar conditions. Lastly, the block parts include four lines, regardless of the meaning, and so they are considered to be similar blocks. To conclude, two different code segments, in Table 3, can be considered similar code segments in this paper. Similar code segments can be codified after the identifying process.

3.2 Codifying Process

The aim of the codifying process is to increase the number of standard forms among students' solutions. Students' program codes are naturally different from each other. The rest of this section discusses the details of the codifying processes.

This process refers to Process 3 in Figure 1. In this process, similar code segments can be codified using rules. These rules can arrange the order of arguments in the condition part and the order of block lines of the similar code segments. For example, in Table 3, the order of conditions of similar code segments are different from each other. After the codifying the condition parts of similar code segments, the condition part of similar code segment-2 can be identical with the condition part of similar code segment-1. Additionally, the order of block parts of similar code segments can be also identical after the codifying process. Then, both similar code segment-1 and similar code segment-2 from Table 3 can be called component A which is illustrated in Table 4.

Table 4: Component A.

Component A

`if ((x = 0)and(t = 0)){`
`y = x + 1`
`z = t - 1`
`print 'y is positive'`
`print 'z is negative'}`

After the codifying process, there will be a transformation from the two similar code segments into the component A. The similar code segments can be seen in the Table 3 while component A is illustrated in the Table 4. Thus, the two similar code segments from the Table 3 can be codified through this process.

3.3 Grouping Process

String matching is the main part of the grouping process. The result of the string match directly affects the group numbers. After the codifying

process, the components can be grouped by the system. This process refers to Process 4 in Figure 1. In this process, each different component can be grouped in terms of component structure in this process. The component structure can be an 'if', 'else-if' or 'else' structure. For example, component A refers to the 'if' structure in Table 4. Moreover, if component A is used by 25 students, all of them can be put into same group. That is, the created group includes 25 components A according to this example.

In addition to this, a whole code script may include not only 'if' structures but also sequence part of code segments, loops, functions etc. That is, a component could represent a sequence part of a code segment, control structure, loop or function etc. Each of them can be put into required groups through the grouping process. In this case, more complex assignments can also be assessed using the approach taken by this research. However, the grouping process should be applied systematically. For instance, a group may include different components which are not identical between each other. Thus, the grouping process is very important in this approach.

3.4 Marking Process

In this process, the examiner needs to comment and mark only one component from each group, rather than all of the components and the rest of the components from each group can be automatically marked by the system using the examiner comments. This process refers to Process 5 in Figure 1. That is, each component from same group can get same comments through this process. Table 5 shows the two programs' code showing the same components as letters.

Table 5: Two program code.

Code-1	Code-2
A	B
B	A
C	C

For example, in Table 5, after the examiner comments component A from Code-1, another component A can be automatically commented in Code-2. In addition to this, we envisage that, in the eventual tool each commented component could be illustrated with different colours. For example, all components A could be highlighted after the examiner comments it (showing that those areas can be ignored when reading other students code as they have already been commented upon).

At the end of the marking process, the examiner can also comment the non-codified similar code segments. This process refers to Process 6 in Figure 1. Thus, each student's code script can be marked and commented by the examiner through the proposed semi-automatic assessment approach.

4 FEASIBILITY OF THE APPROACH

This section gives information on data collection, and the analysis and discussion of parts of the case study. The following sub-section explains the data collection.

4.1 Data Collection

Data was collected to ensure the feasibility of the approach proposed in this research. A question on 'else-if' control structures was asked to students taking semester one (2014) of the introduction to programming module at Loughborough University. The lab exam, which 53 students attempted, asked about the usage of 'else-if' structures. 51 of the 53 students tried to use an 'else-if' structure; in other words, 96% of students used 'else-if' structures in their solutions. The students used the Python programming language to complete the task, which was as follows:

> Write a program which asks the user to enter integer values for the radius of the base and the height of a cone. The program should then calculate the surface area and the volume of the cone, and the program should check for the following conditions before calculating the area:
> If the radius or height is not a positive number then the program should print the message:
> ERROR: Both the r and h values must be positive numbers!
> If the radius or the height is more than 100 then the program should print the message:
> ERROR: Both the r and h values must be less than or equal to 100 cm!
> You MUST utilise the 'if...elif…else' statement in your code.

According to the question, students must use the 'else-if' structure in his/her program code. Each requirement needs to be highlighted in the questions such as variable names, print messages, control structures etc.

4.2 Analysis and Discussion

Students' code segments were manually identified,

codified, grouped and marked in this research. The analysis section assumes that code segments written by students are similar code segments. This process refers to Process 2 in Figure 1. They have been analysed in terms of 'if' control structures which are 'if', 'else-if' and 'else' structures. Table 6 shows the three real source code which were written by the students.

Table 6: Real similar code segments of students.

Name	Students' Similar Code Segments
X	`if (h<=0 or r<=0):` `  print "r and h must be positive"`
	`elif (h>100 or r>100):` `  print "r and h less than 100"`

	`else:`
	`  s=3.14*r(r+(h*h+r*r)**0.5)`
	`  v=(3.14*r*r*h)/3`
	`  print s, "and", v`

Y	`if (h<=0 or r<=0):` `  print "r and h must be positive"`
Z	`if (h>100 or r>100):` `  print "r and h less than 100"`

	`else:`
	`  s=3.14*r(r+(h*h+r*r)**0.5)`
	`  v=(3.14*r*r*h)/3`
	`  print s, "and", v`

The X refers to the 'else-if' structure; Y refers to the 'if' structure, and lastly, Z refers to the 'else' structure in Table 6. Initially, each of them is manually codified. This process refers to Process 3 in Figure 1. In this process, the orders of arguments in condition parts of theirs and the orders of block lines of theirs were fixed manually. Then, they were considered component X, Y and Z after the codifying process. Table 7 gives information on the numbers of components which were obtained from the codifying process.

Table 7: Number of components.

Component Name	Number of Components
X (else-if structure)	43
Y (if structure)	1
Z (else structure)	1

According to Table 7, 43 students used component X, and one student used both components Y and Z (out of a total of 53 students). That is, 44 students' similar code segments were codified using this approach. For the remaining nine students' similar code segments were not codified because their segments did not resemble the code structures X, Y and Z. Eight of these nine students tried to use 'else-if' structures without else statements. The remaining one of these nine students used nested structures. Thus, these nine students' code segments would have to be marked and commented on by the examiner because they were not codified, which is shown as Process 6 in Figure 1. Then, the component X, Y and Z were put into groups. This process refers to Process 4 in Figure 1. Table 8 shows the groups of components.

Table 8: Groups of components.

Group Name	Group1	Group2	Group3
Component Name	X	Y	Z

Group 1 includes component X, Group 2 includes component Y and Group 3 includes component Z. After the grouping process, only one components from the each group needs to be marked and commented by the examiner. Then, the rest of the non-commented components form each group need to be assessed by the system. This process refers to Process 5 in Figure 1. Table 9 shows the components assessed by the examiner using the proposed approach.

Table 9: Numbers of components marked by examiner and proposed approach.

Component Name	Component Number	Assessed by the Examiner	Assessed by the Proposed Approach
X	43	1	42
Y	1	1	0
Z	1	1	0

In Table 9, three of 45 components need to be marked by the examiner which refers to 7% of components. The rest of the components, 42 of the 45 components which refer to 93% of components, can be assessed by the proposed approach.

At the end of the assessment process, only the three different components and the nine non-codified similar code segments need to be marked by the examiner. These highlighted numbers are only related to the introduced example in this paper. However, their numbers can change due to certain issues. The next section will discuss issues identified with the approach outlined.

5 ISSUES WITH THE PROPOSED SEMI-AUTOMATIC ASSESSMENT APPROACH

Although we successfully analysed the short solutions of novice students, we did encounter some problems. For example, different variable names, print messages and order of components are a problem for this approach. These problems are discussed below.

5.1 Problem 1: Use of Different Variable Names

Table 10 shows two different similar code segments. 'a' and 'area' are variable names in Table 10.

Table 10: Different similar code segments.

No	Code
1	`if(a < 0)` `  print "should be positive"`
2	`if(area < 0)` `  print "should be positive"`

Although both statements are meant to make the same comparison, two different components can be created by the codifying process due to the use of different syntax in variable names. That is, this issue can cause the creation of extra, redundant components.

5.2 Problem 2: Use of Different Print Messages

Table 11 shows two different similar code segments. 'Should be positive' and 'should not be negative' are the print message parts of the code.

Table 11: Different similar code segments.

No	Code
1	`if(a < 0)` `  print "should be positive"`
2	`if(a < 0)` `  print "Should not be negative"`

Although both programs give the same message in terms of meaning, their wording (i.e. strings) is very different. Thus, in the codifying process, they are codified differently; two different components can be created due to the use of different syntax in print messages.

Both problem 1 and problem 2 can cause the creation of a new component in the assessment process. If these issues are solved, component numbers and also group numbers do not increase redundantly. They can be solved through a user interface. For example, it can be designed such as the Scratch programming user interface (Resnick et al., 2009). Students can drag and drop each part of the code segment and so students will use same variable names and print messages.

5.3 Problem 3: Order of Components

The differing order of components could be another issue for this approach. Table 12 shows two different programming codes showing components as a letter.

Table 12: Two different programming code.

Code-1	Code-2
A	B
B	A
C	C
E	D

In Table 12, two different program codes include components which are A, B, C, D and E. The orders of components in the students' code are different from each other. In this case, the automatically commented components can be commented incorrectly due to different orders of the components. That is, students may get some incorrect feedback because the incorrect order of the components can cause logical errors. Thus, the different order of components is a recognised issue for this approach which needs to be addressed in the future.

6 CONCLUSIONS

This research paper has introduced a semi-automatic assessment approach which helps examiners by reducing the number of components to be marked. It was applied manually and the initial results are very encouraging. This approach has four important parts: identifying, codifying, grouping and marking. Codifying the similar code segments is also a challenging part of it. The similarity measurement of code segments is also discussed briefly, though outlining this further will be the role of another paper due to space constraints. Potential issues relating to the application of this approach are also discussed. To conclude, the proposed approach is feasible according to the result of the case study because the examiner only needed to assess three components from 45 components. That is, 7% of components were assessed by the examiner. Thus,

examiner workloads can be partially reduced and consistent feedback can be provided through the proposed semi-automatic assessment approach.

6.1 Further Work

The approach outlined focuses primarily on 'if' structures. However, in order to provide a useful tool, additional control structures will need to be supported. The first step in this will be support for 'for' loops, as these are common across all high-level programming languages. Similarity measurement is also very important in order to identify similar code segments in different students' code. This particular area will be developed further in the future. In terms of tool support to enable real world testing of the approach a drag and drop user interface is under development at the present time.

REFERENCES

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

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

Clark, I. (2011). Formative Assessment: Policy, Perspectives and Practice. *Florida Journal of Educational Administration & Policy*, 4(2), 158–180.

Foxley, E., Higgins, C., Hegazy, T., Symeonidis, P. & Tsintsifas, A. (2001). The CourseMaster CBA system: Improvements over Ceilidh.

Jackson, D. (2000). A semi-automated approach to online assessment. *ACM SIGCSE Bulletin* ACM, 164.

Joy, M., Griffiths, N. & Boyatt, R. (2005). The boss online submission and assessment system. *Journal on Educational Resources in Computing* (JERIC), 5(3), 2.

Kaczmarczyk, L.C., Petrick, E.R., East, J.P. & Herman, G.L. (2010). "Identifying student misconceptions of programming", *Proceedings of the 41st ACM technical symposium on Computer science educationACM*, pp. 107.

Meerbaum-Salant, O., Armoni, M. & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education*, 23(3), 239-264.

Melmer, R., Burmaster, E., & James, T. K. (2008). Attributes of effective formative assessment. *Washington, DC: Council of Chief State School Officers. Retrieved October*, 7, 2008.

Palmer, J., Williams, R., & Dreher, H. (2002). Automated essay grading system applied to a first year university subject – How can we do it better?. In *Proceedings of Informing Science 2002 Conference, Cork, Ireland, June* (pp. 19-21).

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K. & Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM,* 52(11), 60-67.

Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C. & Velázquez-Iturbide, Á. (2014). Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31, 453-460.

Saikkonen, R., Malmi, L. & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *ACM Sigcse Bulletin ACM*, pp. 133.

Schmidt, D.A. (2012). [Online] Programming Language Semantics. *Kansas State University*. Available from: http://people.cis.ksu.edu/~schmidt/705a/Lectures/chapter.pdf [Accessed 02 September 2015]

Scriven, M.S. (1967). The methodology of evaluation (Perspectives of Curriculum Evaluation, and AERA monograph Series on Curriculum Evaluation, No. 1. *Chicago: Rand NcNally.*

Sharma, K., Banerjee, K., Vikas, I. & Mandal, C. (2014). Automated checking of the violation of precedence of conditions in else-if constructs in students' programs. *MOOC, Innovation and Technology in Education (MITE), 2014 IEEE International Conference on* IEEE, 201.

Soloway, E. & Spohrer, J. C. (2013). *Studying the novice programmer.* Psychology Press.

Wang, T., Su, X., Wang, Y. & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2), 99-107.