A New Similarity-based Method for Assessing Programming Assignments using Symbolic Execution

Sara Mernissi Arifi*, Rachid Ben Abboub and Azeddine Zahić

Intelligent Systems and Applications Laboratory, Sidi Mohamed Ben Abdellah University (USMBA), Fez, Morocco.

Abstract
Automated assessment of programming assignments continues to be an attractive research topic in computer science education. Various methods have been contrived and applied for over many years. They aim at improving students’ programming skills and optimizing the teaching staff time. Despite the various works carried out in this context, there still gaps to be addressed. The major shortcoming is related to the diversity of correct programs that resolve a programming problem. In this paper, we propose a new evaluation method that can well support the grading process of students’ programming assignments. The proposed method consists of measuring the gap between the student-provided program and the teacher-provided models by using a semantic similarity measure. Our proposed similarity measure is based on the symbolic execution technique that compares two programs according to their symbolic execution outputs. In this work, we present two variants of this method. The first one compares semantically the evaluated program to a single teacher-provided model. Experiment of this variant gave satisfactory results. However, it seems to be unsuitable when the evaluated program and the teacher-provided model are based on completely different algorithms. So, we proposed a second variant as an improvement of our method, which consists of providing, not all the solution programs, but one representative model for each group of completely different solutions to form the solution plan of a programming problem. Experiencing the method variant based on the multi-models solution plan showed enhanced performing compared to the one based on a single model solution plan.

Keywords. Program automated evaluation; static analysis; non-structural analysis; semantic similarity; symbolic execution.

INTRODUCTION
Among various aspects of education, the evaluation of the learning outcomes is the most critical task for ensuring good pedagogical quality. Here, the aim is to provide a measure of learners or students’ comprehension and performance and assist both the educators and the learners to get a deeper insight of their knowledge level and gaps (Jeremić, Jovanović, & Gašević, 2012)(VanLehn, 2008). Evaluation of learning is considered as a systematic process dedicated to monitor the success of a course according to the achievement of the intended learning outcomes. Three main tasks can be performed during this process: grading students’ assignments, giving feedback for both students and educators, and detecting plagiarism in the students’ answers. Through evaluation, educators make the judgment about reaching their goals in order to review, adjust and improve their pedagogical methods. Moreover, the undertaken evaluation can give more motivation for students to improve, increase and develop their learning.

Computer science subjects, such as algorithms and programming, databases design and systems, web technologies, and artificial intelligence techniques, are essential, not only in computer science curriculum, but also in all the branches of technical and engineering education. Over the last decade, computer programming competencies were among the most sought skills in many job profiles. Consequently, the number of students enrolled in universities in programming courses has increased, and new teaching approaches have emerged for exploring new ways to boost learning programming techniques and languages, such as collaborative learning, project-based learning (PBL), e-Learning, m-Learning and massive open online courses (MOOCs). Moreover, computer programming courses, whether in the classroom or through the web, are challenging because they are very practical and problem-solving oriented. Indeed, the practice constitutes an effective tool for such courses, especially for introductory classes, to achieve programming courses goals. Many studies approved the existence of a positive correlation between the amount of practice and the improvement of the students’ skills. They also confirm that, the more students are faced with various programming problems, the more their programming and problem-solving skills are refined.

In this paper, we are concerned with the evaluation in the context of computer programming courses. Academics and software professionals stress different attributes to measure the goodness of programs such as correctness, efficiency and maintainability (Joy, Griffiths, & Boyatt, 2005);(Moha, Gueheneuc, Duchien, & Le Meur, 2010)). We focus our study on grading programming assignments in introductory classes according to the correctness of programs rather than their efficiency or maintainability. We try to check whether the functionality and the design of the submitted program conform to the programming problem requirements.

The huge number of programming assignments that have to be graded and provided with feedback in a fair and timely manner makes the human grading no more suitable due to time-consuming and much subjectivity. In this respect, automated assessment of programs has emerged as an alternative and a staple approach in computer science courses that can give marks automatically for students, ensuring more
subjectivity and timeliness. Automated grading has many benefits for students and instructors such as improving students’ learning by offering immediate feedback, raising enthusiasm to practice and learn, and reducing substantial workloads for instructors and other resources. Several systems and tools have been developed and used for automated grading of students’ programs for many years ((Jackson & Usher, 1997); (C. A. Higgins, Gray, Symeonidis, & Tsintsilas, 2005); (Helmick, 2007); (Blumenstein, Green, Fogelman, Nguyen, & Muthukkumara-samy, 2008)). They are based on techniques that may be categorized in three main approaches. The first one is *dynamic analysis*, which consists of executing the examined program with a set of test-cases, and then matching the results with the expected outputs. The second approach is *Static analysis*, which evaluates the student’s program without running it. It consists of checking the source code using software metrics like lines of code, number of variables, control statements, control flow, etc., to check the correctness of the student code. The third approach is *Similarity-based analysis* that attempts to evaluate students’ programs by comparing them to a model program provided by the instructor. Here, an abstract representation is first generated for both the assessed program and the model program based on their source codes. Second, the resulted representations are compared using a similarity measure in order to evaluate their similarities. Finally, a grade is assigned to the student program according to the measured similarity degree.

Despite the success of Dynamic and static analysis approaches in software testing, their use in student program grading presents many deficiencies. The use of static analysis to grade a program according to its complexity or to the number of lines of code it is composed of is pointless. Moreover, dynamic analysis is unsuitable for student programs that usually don't compile. In contrast, the similarity-based approach holds great promise in the field of automated grading of student programs. Indeed, using an abstract representation of the source code rather than metrics to analyse a program, allows considering semantic proprieties in program grading. On the other hand, the use of a similarity measure makes the grading process adjustable according to various programming learning goals.

In spite of these advantages, most of the techniques of this approach rely solely on the structure of the programs to derive an evaluation process. In this paper, we present a new similarity-based method that additionally considers the behaviour of the compared programs, in grading student programs. The proposed method uses the symbolic execution approach to capture the behaviour of the compared programs without real execution on a machine (King, 1976). This consists of executing the source code of a program, using symbolic rather than concrete values. Our proposed method operates in three main steps. First, the source code of the examined program and the model programs are transformed into a Control Flow Graph (CFG); the nodes of the CFG represent the basic blocks of instructions, and the edges represent the execution between blocks. Second, a non-structural matching between the representations of the assessed program and the provided model program is performed. For that, we use a semantic similarity measure that compares the symbolic execution outputs of the blocks of each program. This ensures that the matched programs are compared according to their semantics rather than their syntactic aspects. Therefore, two programs could be textually different but semantically equivalent since they have the same symbolic execution results. Finally, a grade is computed and assigned to the submitted program, based on the degree of similarity to the model program. As a prior work (Sara Mernissi Arifi, Zahi, & Benabbou, 2016), we validated our proposed method with real students’ programs that are based on a same algorithm to solve a programming problem. Experiments were conducted using a single model program provided by the teacher over C programs, submitted by novice students. The experimental results showed a considerable correlation to the human grading approach. However, we obtained unsatisfactory results when grading programs based on algorithms completely different from the algorithm adopted in the model program to solve the programming problem. Thus, an incorrect grade is assigned to the student program in such cases. This can be explained by the fact that two programs can be semantically equivalent even if they have different symbolic execution results. Therefore, basing the assessment on a single-model could lead to erroneous grades in some cases. Hence, we extended our proposed evaluation method to a multi-models approach, which consists of comparing the assessed program with various model programs provided by the instructor. Here, each model program represents a range of program solutions based on the same algorithm to resolve the considered programming problem.

The remainder of this paper is organized as follows: section 2 presents related work in the field of automated assessment of programming assignments. Section 3 describes the proposed C programs grading methods using the single and the multi-model evaluation processes. In section 4, we present and discuss the experimental results of the proposed methods.

**RELATED WORK**

Several systems and tools have been developed in the field of automated assessment of programming assignments over the last decades ((Ala-Mutka, 2005); (Douce, Livingstone, & Orwell, 2005); (Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010)). These studies concluded that a grading system should select the features of programming assignments to judge their correctness. In fact, these characteristics are borrowed from software quality field. The focus is placed on the correctness, which measures the compliance to the programming assignment requirements, and indicates if the submitted program produces the desired output. Other features of programming assignments such as maintainability, efficiency, reliability and complexity are less important in introductory courses. In (Ala-Mutka, 2005), the authors classify the features of programming assignments according to the analysis techniques, whether their evaluation requires the execution of the program or can be performed through the program source code. In the present paper, we recognize three main approaches of program analysis.
The first approach is Dynamic analysis. It attempts to evaluate the goodness of the student’s program, according to its behaviour on inputs provided by the instructor. This approach is usually used for evaluating the correctness, the efficiency and the robustness. Dynamic analysis consists of executing the examined program with a battery of test-cases to check whether it produces the expected outputs. Its use in introductory classes usually fails to give fair grades; two main limitations can be distinguished. First, in introductory classes the assessed programs are submitted by novice students; by this fact, they are usually not ready to be executed, neither producing outputs, nor terminating. Second, dynamic analysis is a black boxing method. Then the source code of the assessed program could not be examined to check if the solution meets the requirements and the recommendations of the course. Thus, a program can be considered as correct from the dynamic analysis view even if it does not respect the exercise guidelines. Many automated evaluation systems are based on this method such as CourseMarker (C. a. Higgins, Gray, Symeonidis, & Tsintsifas, 2005) where the program profile is constrained by measuring its functionality and attributes in order to be graded.

The second approach to automated grading of programs is static analysis. Unlike the dynamic analysis, it consists of gathering information about the source code of the submitted program without executing it. It is supported by source code analysis techniques, used to detect deficiencies in the assessed source code. Thus, static analysis can be used to evaluate the complexity, the maintainability, and the correctness by locating errors that have been left unnoticed or are not covered when using a limited set of test-cases. Source code analysis aims to characterize the submitted programs, by indicators calculated using software metrics such as the number of lines of code; the probability of execution error occurrence that comes from the use of dynamic structures; and the complexity by McCabe metrics. Besides, programming style analysis, which informs about the source code comprehensibility, can also be checked by metrics like comment density, descriptive variables names, indentation and modular programming. The advantage of using metrics is that they are easy to calculate, but then again, the semantics of a program cannot be analysed to see how well the student deals with the programming task, therefore the evaluation results could be unfair.

The last approach is similarity-based analysis which consists of evaluating a student's program by comparing it to a model program provided by the instructor. The grade is then assigned according to the degree of similarity between the examined program and the model program. Several techniques of this approach have been proposed and used. They can be characterized by a generic process that is deployed in three main steps. First, both the student’s program and the model program are transformed into intermediate representations such as pseudo-code, trees or graphs. The aim of such abstract representation is to reduce the syntactic variations between the compared programs. Second, the obtained representations are compared using a similarity measure in order to calculate the degree of resemblance to the model program. Here, several aspects like size, statements, structure, etc. can be considered in the measurement. Finally, a mark is assigned according to the obtained similarity degree. Techniques of this approach can be distinguished according to the representation used to abstract the compared programs, among them techniques that are based on software metrics. The idea is to characterize the student’s program and the model program by a set of metrics calculated from their source code. Then a grade is derived according to the obtained values. QUIMERA (Fonte, Boas, Da Cruz, Gancarski, & Henrique, 2012) adopts this technique as a part of a fine programming assessment process. Other techniques propose to convert both the student program and the model program into pseudo-codes to perform a structural comparison. For this purpose, the compared programs source codes are tokenized using parsers or lexical analysers, and then transformed into an XML representation of program abstract syntax trees (AST). This abstract representation allows showing the basic algorithmic structure of the programs. Environment for Learning Programming (ELP) (Truong, Roe, & Bancroft, 2004) and (Khirulnuzam, Ahmad, & Nordin, 2007) adopt this approach to decide whether the student program and the model program are similar; and compute the similarity degree between the associated ASTs to grade the student program. In addition, the authors of (Khirulnuzam et al., 2007) propose to use several different model programs rather single one, trying to consider all the possible solutions in the assessment. The graph structure is also quite often used for representing the compared programs. In (Wang, Su, Wang, & Ma, 2007) the student program and the model program are firstly transformed into a system dependence graph (SDG) which incorporates control dependence and data dependence into a single structure. In an SDG, nodes represent statements of the program, and edges represent dependencies among nodes like control dependence, flow dependence, declaration dependence and function-call. Then a standardization process is performed through semantic-preserving transformations, in order to eliminate the syntactic variations. Secondly, the standardized student SDG and the standardized model SDG are matched according to three levels: structure, size and statements, to calculate their semantic similarity degree. Finally, a mark is given based on the obtained similarity degrees. The refereed work, address also the problem of the multitude of solutions which resolve differently the same programming problem. In (Naudé, Greyling, & Vogts, 2010), the authors propose to represent the evaluated program by a graph constructed in three steps. First, the source code is transformed into an abstract syntax tree. Normalization is then applied to the obtained tree in order to eliminate diversities such as algebraic identities and the choice of identifiers. Finally, the normalized tree is transformed into a graph by adding dependencies between statements, as well as use-definition relationships. Moreover, a graph similarity measure relying on the principle that similar vertices should have similar neighbours is proposed. It is used to derive a grade for a submitted student program, by comparing it to the already marked solutions. Similarity-based approach is promising for automated grading of programming assignment compared with dynamic and static analysis methods. Semantic comparison makes the automated grading process closer to the instructor’s one.
Thus, the focus can be held on specific programming tasks such as practicing programming concepts, algorithms and data structures. Reviewed tools which adopt the similarity-based approach to grade a programming assignment are subject to limits in three main issues. Firstly, some systems perform a structural matching between the assessed and the model programs; hence the provided grade reflects the structure rather than the behaviour of the assessed program. Secondly, the main issue in the field of program grading which is the multitude of models that could form the solution plan of a programming problem is treated through the standardization or semantic preserving transforming of the compared programs, and thirdly, the grade calculation is not parametrised in the way to allow the instructor to reflect his own requirements through a customized grading. In this respect, we propose a new similarity-based grading method, based on symbolic execution technic to analyse the compared programs to check the behaviour of the assessed program and deal with the issue related to the multitude of model-programs. The adopted similarity measurement is performed through a weighted formula to express the instructor’s requirements.

THE PROPOSED METHOD

Our proposal for the automated grading of programming assignments issue is a new similarity-based method that improves the general process of the similarity-based approach through the following performances.

- The ability of the analysis process to consider the behavior of the evaluated programs, in the similarity measurement, against the model program behavior. Here, we propose the use of the symbolic execution approach that executes a program with symbolic values rather than concrete values. In this way, we can characterize the behavior of a given program, according to its symbolic outputs without executing it. Thus, the evaluated program is graded according to its similarity degree to a model program. The comparison is performed in terms of semantic similarity, measured using symbolic execution outputs of both programs. (Sara Mernissi Arifi, Zahi, & Benabbou, 2016)

- The ability of the analysis process to reduce the diversities between the compared programs; two levels of diversity are distinguished here. The first level concerns the solutions programs, resolving the same programming problem, according to the same algorithm or design. In fact, the student programs and the model program can express the same solution yet producing different source codes (different variables names, different loops, etc.). Here, we propose the use of CFG to abstract the compared programs. The second level concerns the programs, resolving the same problem by different algorithms or designs. Here, the evaluation process considers multiple model programs, where each model program represents a class of student programs that resolves the problem according to the same algorithm.

- The ability to provide the student with feedback made of the comments attributed if necessary by the evaluator to each node in the constituted CFG. Hence the grade automatically produced is justified and the student could exploit the provided feedback to figure out the committed errors.

Figure 1 shows the process of the proposed method. It is deployed in three main steps:

- **Representation of programs** generates the CFG of the model program and the student program and provides a semantic representation using their symbolic execution results.

- **Similarity evaluation** matches the representations of the compared programs using the proposed similarity measure in order to derive the similarity degree.

- **Grading** assigns a grade to the student program according to its similarity degree to the model program.

![Symbolic execution based grading method](image)

**Figure 1.** Symbolic execution based grading method

**Representation of programs using symbolic execution**

Symbolic execution is a program analysis technique introduced by King in the mid ’70s in the context of software testing to check whether a certain property can be violated by a program (King, 1976). It has been applied to various software engineering tasks, such as test-case generation, bug finding, software verification, program debugging (Qi, Roychoudhury, Liang, & Vaswani, 2009); (Artzi, Dolby, Tip, & Pistoia, 2010); (Ge, Taneja, Xie, & Tillmann, 2011); (Li, Chen, Wang, & Xu, 2013), discovery of invariants (Csallner, Tillmann, & Smaragdakis, 2008); (Zhang, Yang, Rungta, Person, & Khurshid, 2014) and worst-case execution time estimation for real-time software (Ghiduk & Ghiduk, 2016), etc. Symbolic execution consists of interpreting the behavior of a program on symbolic values rather than concrete values. More precisely, the values assigned to the variables along the program execution are represented by algebraic expressions of
symbolic values. A symbolic value is defined by a symbol, which is usually noted by a Greek letter, and a set of concrete values it can range over. The process of symbolic execution of a program maintains the program state through the execution by producing two elements: variables store and path conditions.

- The variables store, noted by the symbol $\delta$, is the result of the mapping from variables to values. It is a set of symbolic expressions that represent the values of the variables along the program execution. For instance, considering the block $B$ composed of two instructions $x=2$; $y = x+3$; and assuming that the variables $x$ and $y$ are respectively associated with the symbolic values $a_1$ and $a_2$. The execution of the block $B$ produces the variable stores $\delta = \{a_1 = 2, a_2 = a_1 + 3\}$.

- The path conditions, noted by $PC$, express the constraints that must be satisfied in order to allow the execution of a path (Baldoni et al., 2016). Each constraint is represented by a conditional expression of symbolic values, such as $\alpha > 1$ or $2 \leq \alpha < \beta$. These constraints are constructed from if-else and loop statements. For instance, considering the block $B$ composed of the instructions $x=0$; $y = x+2$; if ($y<10$ && $y \geq x$) $x=0$; and assuming that the variables $x$ and $y$ are respectively associated with the symbolic values $a_1$ and $a_2$. The execution of the block $B$ produces the path conditions $PC = \{a_1 \leq a_2 < 10\}$; this means that if $PC$ is satisfied, then the path is executed.

Hence, the student program and the model program are represented by their symbolic execution outputs expressed in term of variable stores and path conditions. To obtain such symbolic representation for a given program, we perform the following steps:

- First, the program is partitioned into a set of basic blocks of instructions. A block is a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. The obtained blocks are then sketched into a Control Flow Graph (CFG) to represent the structure of the program. The nodes of the CFG are the basic blocks and the arcs connect the basic blocks to visualize all the possible execution paths.

- Second, symbolic execution is performed over the program CFG in order to characterize its behaviour. For each basic block in the CFG we maintain the variable stores and the path conditions.

Figure 2 illustrates the process of representing a program by its symbolic execution outputs. We consider a simple C program and then we construct the associated CFG visualizing its different basic blocks. Finally, we present the results of the symbolic execution expressed in term of variable stores and path conditions, obtained for each basic block; $\delta$ designate symbolic values of the variables and $PC$ represents the path condition.

![Figure 2. Symbolic execution process for a simple C program.](image-url)
Semantic similarity measure

By using symbolic execution to characterize the student program and the model program, we can capture their semantics, and then overcome the diversities occurring due to the multitude of programs that solve the same problem. Definitely, two programs can be textually different but semantically equivalent, since they have the same symbolic execution outputs. Hence, based on this assumption, we propose a new similarity measure that compares two programs according to their symbolic execution outputs, namely variable stores and path conditions. Using this similarity measure, the student program is matched to a model program representing all the potential solutions to the programming problem. The proposed measure relies on similarity between the basic blocks. We can note that when symbolic execution of two basic blocks leads to the same variable stores, this means that they are producing the same outputs when using the same inputs. Also, when two basic blocks have the same path conditions during their symbolic execution, this means that the same conditions have been satisfied to execute these basic blocks. Hence, we can say that these two basic blocks are semantically equivalent when having the same variable stores and path conditions as a result of their symbolic execution. In other words, these basic blocks behave in the same way when they are executed in the same data and conditions context. Figure 3 summarizes the comparison process of two programs using symbolic execution results.

Consider a student program $P_s$ and the model program $P_m$, where $P_s$ is described by the set of basic blocks $B_{s,i}:1 \leq i \leq n_s$, and $P_m$ is described by a set of basic blocks $B_{m,j}:1 \leq j \leq n_m$. The similarity between $P_s$ and $P_m$ is evaluated in two main steps:

- **Block similarity evaluation** consists to evaluate the similarity between $P_s$ and $P_m$ at the block level. For each basic block $B_{m,j}$ of $P_m$, we compute the similarity degree by sequentially matching $j$ to all the basic blocks belonging to $P_s$. Each comparison generates two measures; the first one represents the similarity degree towards the data store, the second one indicates the similarity degree concerning the path conditions. Both measures contribute equally to the composition of the block similarity degree of the compared basic blocks. Through all the performed comparisons, the maximal similarity degree is retained for $B_{m,j}$. Consequently, the block similarity degree for $B_{m,j}$ is computed by the following formula:

$$Sim_{block}(B_{m,j}) = \max_{1 \leq i \leq n_s} \left( S_{s,i} + S_{pc,i} \right) / 2$$  

Where:

- $S_{s,i}$ is the similarity degree of the variables’ stores of the matched basic blocks $B_{s,i}$ and $B_{m,j}$
- $S_{pc,i}$ is the similarity degree of the path conditions of the matched blocks $B_{s,i}$ and $B_{m,j}$
- $n_s$ is the number of basic blocks in $P_s$

- **Global similarity evaluation**: is calculated using the block similarity degrees carried out in the previous step. For that, we adopt a weighting-based aggregation technique, which considers the basic blocks of the model program with different importance in the overall evaluation. The weights are assigned to the basic blocks by the instructor, according to his own requirements and objectives. In the case of considering the basic blocks in

![Figure 3. Semantic similarity measurement process using symbolic execution outputs](image-url)
the same level, we assign all weights to 1. Over the basic blocks weighting, the instructor can shape the grade calculation according to his own requirements and the behaviours he is expecting from the student program. In fact, each basic block contributes proportionally to its weight in the similarity evaluation. Hence, the global similarity between $P_e$ and $P_m$ is calculated according to the following formula:

\[
WSim(P_e, P_m) = \sum_{k=1}^{nm} \frac{Sim_{block}(B_{m,k}) \cdot \text{weight}(B_{m,k})}{nm}
\]  

Where:
- $Sim_{block}(B_{m,k})$ is the block similarity degree generated through all possible comparisons with the basic block $B_{m,k}$.
- $nm$ is the number of the basic blocks in the model

The basic blocs weighting could be as well exploited in the feedback generation. Hence the order in listing the generated feed-back takes into consideration the weights assigned to the model basic blocks.

To illustrate the semantic similarity measurement, we consider the example presented in figure 4. It shows the student program “Pe.c” and the program model “Pm.c”, with their CFG and symbolic execution outputs obtained for each basic block. There are three basic blocks for each program; we note block 1, 2 and 3 for the student program, and block A, B and C for the program model. We remark that the programs “Pe.c” and “Pm.c” have identical symbolic execution outputs. So, we can say that they are semantically equivalent even if they make use of different loops and variables.

![Figure 4. Example of similarity measurement between two C programs](image-url)
Moreover, the similarity between the student program and the model program is quantified by a value within the interval [0, 1]; “0” in the case of totally different programs, and “1” in the case of totally similar programs. This measure serves as a basis for calculating the grade attributed to the student program, according to its total or partial conformity to the instructor’s requirements.

Multi-model grading

The use of our single-provided model program method when grading a student program, certainly deals with the major limits of the static analysis method, and avoid the matching of the assessed program to all the possible solutions. However, the assessment could lead to erroneous grades when the submitted program uses a different algorithm against the model program, yet solving the same programming problem. This is since two programs can be semantically equivalent even if they have different symbolic execution outputs. In order of a further clarification of this point, we consider the example of the programming problem where the student has to write a C program that switches the values of two variables. Figure 5 shows the model program used to grade the student program submitted for this problem. In the model program a temporary variable is used to swap the variables while the student program makes use of arithmetic operations to perform the same task. Both the student program and the model program solve the switching problem. However, they don’t have identical symbolic execution outputs. Table 1 shows the results of symbolic execution of the compared programs. We have a single basic block in each program; the path conditions of both basic blocks are the same, but the variable stores are not alike. Therefore, the similarity measurement leads to incorrect grading results.

Table 1: Symbolic execution outputs of two semantically equivalent programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Model program: using a temporary variable to switch variables a and b</th>
<th>Student program: switching the variables a and b without using a temp variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 int a,b,temp;</td>
<td>1 inta,b;</td>
</tr>
<tr>
<td></td>
<td>2 printf(&quot;Enter a :&quot;);</td>
<td>2 printf(&quot;Enter a :&quot;);</td>
</tr>
<tr>
<td></td>
<td>3 scanf(&quot;%d&quot;, &amp;a);</td>
<td>3 scanf(&quot;%d&quot;, &amp;a);</td>
</tr>
<tr>
<td></td>
<td>4 printf(&quot;Enter b :&quot;);</td>
<td>4 printf(&quot;Enter b :&quot;);</td>
</tr>
<tr>
<td></td>
<td>5 scanf(&quot;%d&quot;, &amp;b);</td>
<td>5 scanf(&quot;%d&quot;, &amp;b);</td>
</tr>
<tr>
<td></td>
<td>6 temp=a;</td>
<td>6 a=a+b;</td>
</tr>
<tr>
<td></td>
<td>7 a=b;</td>
<td>7 b=a-b;</td>
</tr>
<tr>
<td></td>
<td>8 b=temp;</td>
<td>8 a=a-b;</td>
</tr>
<tr>
<td></td>
<td>9 printf(&quot;a= %d and b= %d&quot;,a,b);</td>
<td>9 printf(&quot;a= %d and b= %d&quot;,a,b);</td>
</tr>
<tr>
<td>Path conditions</td>
<td>PC : true</td>
<td>PC : true</td>
</tr>
<tr>
<td>Variable stores</td>
<td>Statement 1: a=α₁, b=α₂, temp=α₃</td>
<td>Statement 1: a=α₁, b=α₂</td>
</tr>
<tr>
<td></td>
<td>Statement 6: α₁=α₁, α₂=α₂, α₃=α₁</td>
<td>Statement 6: α₁=α₁+ α₂, α₂=α₂</td>
</tr>
<tr>
<td></td>
<td>Statement 7: α₁=α₂, α₂=α₂, α₃=α₁</td>
<td>Statement 7: α₁=α₁+ α₂, α₂=α₂</td>
</tr>
<tr>
<td></td>
<td>Statement 8: α₁=α₂, α₂=α₁, α₃=α₁</td>
<td>Statement 8: α₁=α₁, α₂=α₂</td>
</tr>
<tr>
<td></td>
<td>δ : {α₁=α₂, α₂=α₁, α₃=α₁}</td>
<td>δ : {α₁=α₂, α₂=α₁}</td>
</tr>
</tbody>
</table>

In the light of the results of this example, we can say that using a single-model based evaluation is powerless when dealing with programs based on completely different algorithms. Consequently, we needed to rethink the solution-plan composition provided by the instructor; which should include several possible program solutions for the considered problem. Hence, we extended the single-model based evaluation method to a multi-model based evaluation process.
by including various programs in the solution-plan. The provided model programs represent all categories of possible solutions; each model program represents a range of program solutions based on the same algorithm to the considered programming problem. Figure 5 shows the enhanced evaluation process. The assessed program is first matched to all the model programs using our similarity measure (equation 2) to determine the semantic similarity degree with each model program. Then, the maximum similarity score is retained to mark the student assignment.

Figure 5. Multi-model evaluation process
The following algorithm presents the adopted evaluation process enhanced through continuous experimenting and investigations:

```
Let Pe be a student program
Let M be the solution plan composed of model-programs provided by the human evaluator, and based each on a different algorithm. Then M={Pm1, Pm2,…Pmn}
Let S be the similarity degree calculated using the formula which returns the similarity degree between two programs (equation 2), S is a value between 0 and 1.
Let G be the grade attributed to the student program and Gmax is the maximum grade designated by the evaluator to be attributed to the evaluated program if it is correct.

Begin
G=0
For each Pm in M
S=WSim(Pe,Pm)
If (S = 100%) then
G=Gmax
exit
Else if (S>G) then
G=S*Gmax
End.
```

**EXPERIMENTS**

Through the conducted experiments, we aim to evaluate the accuracy and the reliability of our proposed assessment method and to investigate the following question: how accurately can the proposed evaluation process based on semantic similarity measurement provide exact grades compared to grades given by human assessors? In fact, through consulting teachers who have considerable experience in grading C programs, we noticed that there is no standardized process in manual grading of programming assignments. The usual method consists of considering the algorithm used by the student and detecting syntactic and logical errors in the source code. In addition, manual grading suffers generally from a lack of objectivity which could be caused by tire, favoritism, etc. For that purpose, three human evaluators were involved to manually grade the experimented programs. Thus, automated grading results were then compared to the average of the three evaluators’ grades. Two stages are considered in the conducted experiment:

- Stage 1: The single-model evaluation process.
- Stage 2: The multi-model evaluation process.

**Implementation of the proposed method**

Our proposed method was implemented as a part of a system under development, referred to as C Language Automated Assessment System (CLAAS). The aim of developing CLAAS is to give the opportunity for the students in our university, to practice the theoretical concepts learned during on-site lessons, and to get a customised and instant feedback to help them progress and motivate them to learn. On the other hand, CLAAS is conceived to assess students’ assignments, whether in exams or exercises. Therefore, teachers could spend the time elapsed in manual evaluation in other productive activities; also, CLAAS could be used as a dashboard for the instructor which displays statistical information in order to make accurate estimation of their methods and strategies.

CLAAS is designed as a hybrid system which consists of two main components, the dynamic and the static workers, gathered together to make use of the strengths of both static and dynamic program analysis methods. The dynamic worker is endowed by two main assets. The first one consists of checking the program correctness through its execution using a battery of test-cases (S M Arifi, Abdellah, Zahi, & Benabbou, 2015). Outputs of the execution are matched against those generated by the execution of a model program with the same inputs. The second one is the use of Clang to compile the evaluated programs (Guntli 2011). The present decision was motivated by the precision Clang offers considering the mapping of compilation errors and the detailed reports it produces, compared to other compilers. This assists the student to find the committed error and to fix it while compiling his program before its submission. Since CLAAS is used to assess programs submitted by beginners in programming courses that are mostly error prone, the static worker was designed to achieve the evaluation of student programs without executing them. It implements the semantic similarity technique presented in this paper.

**Evaluation of the single-model based assessment**

For the purpose to check the accuracy of using a single-model based evaluation process, a larger experimentation based on real students’ program was planned through grading 185 students’ programs using CLAAS static worker. The grades produced by the automated evaluation tool were then compared to the grades given by human assessors. Weights included in the adopted semantic similarity measurement formula (equation 2) are assigned by the evaluators to each basic block of the model program in order to tune the grading formula according to their requirements as in manual grading. The choice of the human evaluators was based on the experience they have in teaching C programming courses and evaluating students’ programs in our university. The average of three human evaluators’ grades was considered for each programming exercise to ensure a maximum of objectivity in the grades matching.

Table 2 presents the experienced programming problems and the programs adopted as models for the evaluation.
Table 2: Experienced exercises with model-programs provided by the evaluator

<table>
<thead>
<tr>
<th>Programming problems</th>
<th>Model-programs</th>
</tr>
</thead>
</table>
| **Program to check whether a number is positive or negative**                        | #include<stdio.h>  
int main() {  
int a;  
printf("enter an integer");  
scanf("%d",&a);  
if(a>0)  
  printf("positive number");  
else if(a<0)  
  printf("negative number");  
else  
  printf("null");  
  return 0;  
}                                                                                                                                 |
| **Program to return the number of grades upper than 10 in the five entered grades** | #include<stdio.h>  
int main() {  
int A, i, j;  
j=0;  
for(i=1;i<=5;i++){  
do {  
 printf("Enter a grade
",i);  
scanf("%d",&A);  
while((A<0)||(A>20))  
if (A>=10)  
  j++;  
}  
printf("n there are %d numbers upper than 10",j);  
return 0;  
}                                                                                                                                 |
| **Program to print stars * relatively to the entered number**                       | #include<stdio.h>  
int main() {  
int A, i;  
do{  
 printf("Enter an integer
");  
scanf("%d",&A);  
}while(A<=0);  
for(i=1;i<=A;i++){  
 printf("**");  
 printf("n");  
 }  
return 0;  
}                                                                                                                                 |
| **Program which asks the user to enter a positive number lower than 100 and repeats that while the entry is not valid** | #include<stdio.h>  
int main() {  
int A;  
do{  
 printf("Enter a positive integer lower than 100
");  
scanf("%d",&A);  
}while((A<0)||(A>100));  
printf("Good answer!
");  
return 0;  
}                                                                                                                                 |

Figures 6, 7, 8 and 9 represents the grades attributed to the programs submitted by a class (of two groups) of 51 students, during a test including 4 programming problems.
Figure 6. Exercise 1: a program which asks the user to enter an integer and returns if the entered number is whether positive or negative.

Figure 7. Exercise 2: a program which asks the user to enter 5 grades between 0 and 20 and returns how many grades are upper than 10.

Figure 8. Exercise 3: a program which asks the user to enter an integer and prints a number of stars * relatively to the entered number.
Figure 9. Exercise 4: a program which asks the user to enter a positive number lower than 100 and repeats that while the entry is not valid.

Figure 10. Grades Global tendency

Figure 10 summarises the results of all the evaluated programs for each exercise. The dotted line represents the average of manual assessment grades and the solid line represents the automated evaluation ones.

In order to show the variance between the manual and the automated grading, a grade precision was calculated for each exercise using the following formula:

\[ \text{Grade precision} = (1 - \frac{\text{manual grade} - \text{system grade}}{\text{maximum grade}}) \times 100\% \]  

(3)

Sector graphs in figure 11 presents the grade precision values for the evaluated programs.
Grade precision is a value between 0% and 100%. When a grade precision is up to 100%, this means that the grade of the automated evaluation grade matches the manual evaluation one. The higher the grade precision, the closer are the manual and automatic assessment grades.

Through the plot which represents the evaluation results of exercise 1, we can see that the adopted method gives satisfactory outcomes. 90% of the evaluated exercises have a grade precision that exceeds 90%. The difference between evaluated programs and the model program exist in terms of the used variables and the order of the “IF” branches. Since the analysis is based on semantic similarity measurement through the use of symbolic execution, CLAAS static worker is not sensitive in such cases.

For exercise 2, 3 and 4, 51% of the assessed programs have a grade precision between 90% and 100%. Through a careful examination of the submitted programs, we noticed that:

- When the variable initialization and the condition in loop “For” are not the same in the student and the model program. For e.g., the statements for(i=0; i<n; i++) and for(i=1; i<=n;i++) are not similar for the static worker in CLAAS. This produces a difference between the manual and the automated evaluation grades, especially when the weight affected to the basic bloc containing the loop is high.

- The static worker is also sensitive to differences related to conditions in the “If” or loop statements. For e.g., if (a<x) and if (a<= x) are considered dissimilar, while the human evaluator could be more tolerant in this case.

### Evaluation of the multi-model assessment

Through the use of CLAAS to test and assess a larger variety of programs, we noticed that the tool produced relatively low grades in comparison to the grades that should be assigned in some programming problems having various solutions that are based on different algorithms. Therefore a second experimentation was planned in order to check the accuracy of using the multi-model based evaluation process as an enhancement of CLAAS static worker. We experienced programming problems which students can resolve in completely different ways. Through this experimentation we aimed to demonstrate how significant can be the improvement effected by the use of the multi-model evaluation process in CLAAS static worker. Table 3 presents the test exercises and the model-programs provided to assess each one.
Table 3: Experienced exercises with model-programs provided by the evaluator

<table>
<thead>
<tr>
<th>Programming problems</th>
<th>Model-programs</th>
</tr>
</thead>
</table>
| **Program to multiply two integers** | ```c
#include<stdio.h>
#include <math.h>
int main() {
    int firstNumber, secondNumber, product;
    printf("Enter two numbers: ");
    scanf("%d%d",&firstNumber, secondNumber);
    product=firstNumber*secondNumber;
    printf("Product = %d", product);
    return 0;
}
``` |
| **Program to calculate the absolute value of a number** | ```c
#include<stdio.h>
#include <math.h>
int main() {
    int n, result;
    printf("Enter an integer "n");
    scanf("%d", &n);
    result = abs(n);
    printf("Absolute value of %d = %d", n, result);
    return 0;
}
``` |
| **Program to swap two variables** | ```c
#include<stdio.h>
int main() {
    int a,b,temp;
    printf("Enter a : ");
    scanf("%d", &a);
    printf("Enter b : ");
    scanf("%d", &b);
    temp=a;
    a=b;
    b=temp;
    printf("a= %d and b= %d",a,b);
    return 0;
}
``` |
The experimentation was conducted on 33 real students’ programs assigned to resolve the programming problems cited above. Figures 12, 13 and 14 presents the grades provided by CLAAS using the single and the multi model evaluation processes compared to human evaluation grades.

Figure 14 presents the group averages for the three assessed programming problems to match the single-model, the multi-model and the manual evaluation results.

Figure 12. Grading of the programming problem: a program to multiply two integers

Figure 13. Grading of the programming problem: a program to calculate the absolute value of a number

Figure 14. Grading of the programming problem: a program to swap two variables
The curves representing the three evaluation methods remain parallel over all the programming problems. Even though, we can obviously notice that the curves representing the multi-model and manual evaluation results are roughly superposed. This confirms that the multi-model based grading is very close to the manual grading. Therefore, using the multi-model based evaluation process provides an upgrade to our proposed assessment method using semantic similarity measurement to grade the student program.

CONCLUSION AND FUTURE WORK

In this paper we introduce an automated assessment process based on semantic similarity measurement through the use of the symbolic execution technic. Our contribution is related to the main problem of the static analysis method based on non-structural comparison of programs, which is the variety of programs that can solve the same programming problem. The program grading is performed according to total or partial semantic equivalence between the assessed program and a model program. Experimental results showed a very good agreement between the proposed automatic assessment method and human evaluators. However, single-model based evaluation is powerless in some cases, among them when the evaluated program and the model program are based on completely different algorithms. Consequently, we needed to rethink the solution-plan composition which needs to include programs representing all the possible solutions. Hence, we adopted a multi-model based evaluation process which consists of including various programs in the solution-plan that represent all categories of possible solutions. A second real experimentation was conducted, and satisfactory results were returned.

As a future work, we intend to conduct a larger experimentation of the multi-model grading method for the purpose to test the maximum of programming problems. In addition, we plan to include additional static analysis methods such as key-word analysis method to CLAAS static worker to improve the assessment method through checking the presence of required or not required words in the evaluated program. This intends to meet the human assessment process in which CLAAS static worker verifies, if it is requested by the exercise, if the student has respected the directives about using or not the specified structures.

Moreover, the evaluation method based on semantic similarity presented in this work is of a keen interest in the feedback issue which presents an essential component in the field of automated evaluation of programs. For instance, indicating what portion of the evaluated code is not similar to any bloc from any model in the solution plan gives an important hint for the student to figure out the committed error. This is among different other uses of the presented evaluation method.

REFERENCES


