Assisted learning of C programming through automated program repair and feed-back generation

Sara Mernissi Arifi, Rachid Ben Abbou, Azeddine Zahi
Intelligent Systems & Applications Laboratory, USMBA, Morocco

Article Info

Article history:

Received Jan 27, 2020
Revised Apr 3, 2020
Accepted Apr 18, 2020

Keywords:

Assisted learning
Errors auto-repair
Feed-back
Program assessment

ABSTRACT

Programming courses are among all the current academic curricula for engineering studies. Unfortunately, students often face difficulties already on the basic concepts. Both students and teachers believe that practical sessions and guided learning lead to good outcomes. On the other hand, it is virtually difficult considering the number of students enrolled on programming courses. This paper presents an automated assessment system for programming assignments, based on two different methods: static and dynamic analysis. The presented system aims at providing the student with an ongoing and various feedback delivered according to the category and the recurrence of errors. The system imbeds an automated error repairing feature for the purposes of insuring the assessment process achievement. It operates if the student fails to submit a correct program despite the feed-back provided by the system. In such cases, the system uses a penalty mechanism, customized by the teacher to grade the student's program. Testing the presented automated system, through assessing real students' assignments, showed promising results compared to manual assessment.

Corresponding Author:

Sara Mernissi Arifi,
Intelligent Systems & Applications Laboratory,
USMBA, Morocco.
Email: smernissi@hotmail.com

1. INTRODUCTION

Most science, mathematics, engineering, and technology programs expect from students to acquire programming skills as a part of their curricula. A universal expectation is that the student should learn the process of solving problems in computer science domain through producing correct programs that compile and behave as expected [1]. Lahtinen, Ala-Mutka, & Järvinen [2] perceived in their study that both students and teachers agreed that practical learning situations were the most useful. Immediate feedback during problem solving has proven useful [3]. However, it is virtually difficult due to time constraints and common courses sizes. Automated program assessment tools have considerable advantages through providing timely feed-back [4]. Several tools have emerged in the field of automated program assessment [5]. The review by Caiza [6] presents an interesting overview of the systems developed for automated grading of programming assignments over the last forty years. Recent research is more focused on feed-back generation and is aware of the key part it plays in the student's learning process [7–10]. Two main approaches were adopted to address the feed-back issue in automated assessment of programming assignments.

The first approach is based on the compiler messages, since they are the first feed-back the student receives. However [11] states that novices can struggle to deal with standard compiler messages, which can be vague or not accurately describing the error in their code. Hence, making compiler messages more suiTable for novices was performed in [12]. It consists of rewriting the compiler messages in layman terms and adding more elaboration to them. In [13], a recognizer parses both the submitted source code and the raw compiler messages, to find out the error type. The provided feed-back consists of reporting the syntax error's type that has been recognised, and a version with highlighted corrected lines in the code. The main limitation of this approach is that it does not address the possible inaccuracy of the compiling reports and could fail to provide useful feed-back even with additional elaboration.

The second approach is based on static analysis of uncompilable program code. Watson, Li, & Lau [14] use a database of common errors to generate three kinds of feedback for Java programs. The first is an explicit feed-back made of possible causes linked to the class in which the error was identified in the database. The second is an implicit corrective feed-back which consists of performing replacements of the erroneous token, retrieved from a ranked list of substitutions ordered by least Levenshtein distance, until the elimination of the error, only in the case of a ‘cannot find symbol’ error. The third is a logic level corrective feed-back which consists of suggesting solutions that have structural similarity to the student’s code. Structural similarity is measured using a tree-based approach and the edit-distance algorithm. The main issue of this method is that it can be applied to a limited collection of errors. In addition, the accuracy of feed-back is affected by the accuracy of the comparison method.

Within this paper we present CLAAS, a C Language Automated Assessment System which makes a synergy of enhanced transcriptions of both discussed approaches [15]. Thus, static analysis in CLAAS is based on a semantic similarity measurement which insures satisfactory precision in the field of programs’ similarity detection [16]. Semantic similarity is used in CLAAS to provide a feed-back that consists of tracking the erroneous or missing parts in the code and assists the student in writing a complete and a good quality program. On the other hand, dynamic analysis in CLAAS is based on the compiler reports, errors categorization, but also on the error recurrence. This makes the provided feed-back progressive and continuous through all the assessment process to address compiling errors. Besides the various feed-back generated through static and dynamic analysis, a correction unit was implemented in CLAAS to prevent students’ discouragement towards persistent compiling errors [17]. This ensures the achievement of the program assessment process and so the feed-back generation through further assessment stages.

2. APPROACH

Figure 1 presents the overall architecture of the proposed automated assessment system. A hybrid system based on two different analysis methods to assist students through a various and elaborate feedback.

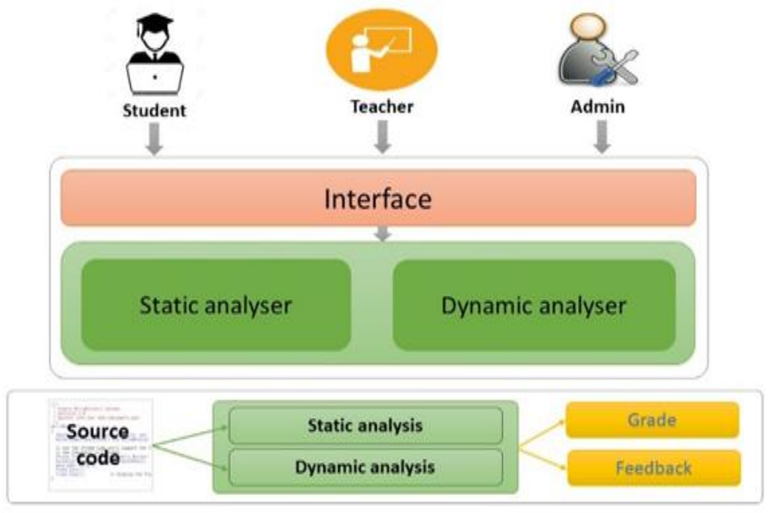


Figure 1. CLAAS architecture

2.1. Feedback through dynamic analysis of programs in CLAAS

Dynamic analysis of programs consists of executing the assessed program using a set of test-cases composed each of input data and the expected outputs to check the program behavior in different contexts. Outputs matching in CLAAS is performed using regular expressions to avoid textual comparison. In addition, the grading formula is customizable through assigning different weights by the teacher to the test-cases to express the marking scheme of the exercise. The enhanced dynamic analysis method adopted in CLAAS in comparison to other program assessment systems was discussed in a previous work [15]. Dynamic analysis method for program automated assessment as shown in Figure 2.

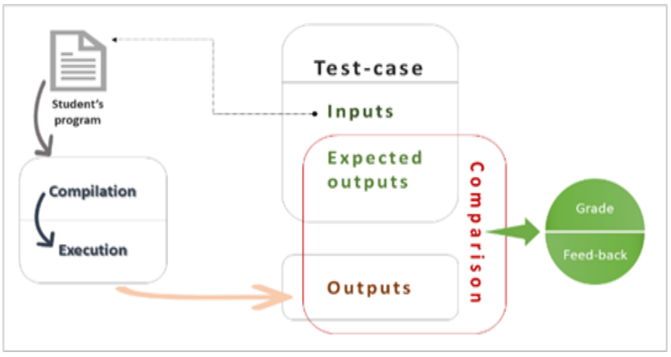


Figure 2. Dynamic analysis method for program automated assessment

2.1.1. Feed-back in the case of compilation failure

The compiler messages are the first automated feed-back received by the student. When students face compiling errors, they often request the help of the teacher or the assistant [18]. Although Debugging is a key programming skill, many novices can struggle with ambiguous and unspecific compiler messages due to their lack of experience in programming field [19].

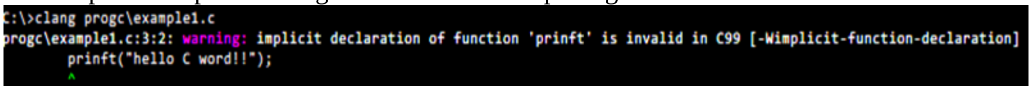
When dynamically analyzing a program, the system returns a progressive feed-back according to the type and the recurrence of a compiling error. Therefore, a base of common errors made by novice programmers in C language learning was designed and fed through manual and automated assessment of real student's programs using CLAAS [17]. The collected errors were clustered within different categories presented in Table 1.

Table 1. Examples of categories in the error base of CLAAS

Error category	Examples of errors
Misspelling errors	Errors in the names of the language functions
Symbol errors	A missed or added semi-colon, a missed & in scanf, missed : after case, etc.
Casting errors	Type conversion
Header errors	A missed #include<<stdio.h>>, etc.
Operators errors	& instead of &&, = instead of ++, etc.
Variable errors	Undeclared or uninitialized variable, etc.
Loop errors	Error in a loop structure.
Control errors	Missing break in switch, etc.

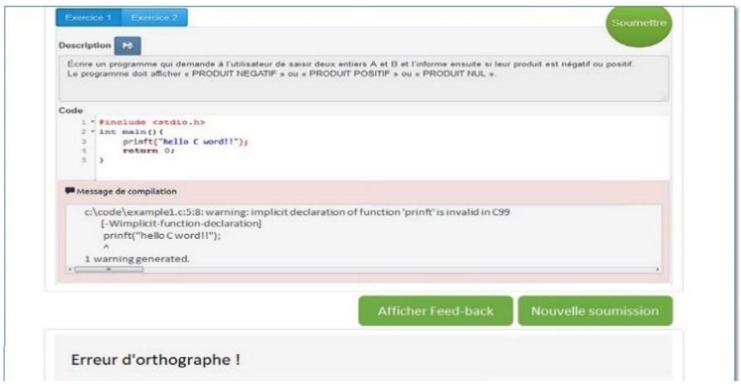
To ensure progressivity in feed-back generation, the evaluator assigns a threshold, and a couple of feedbacks for each category. The first one is delivered when the number of occurrence of errors belonging to the same category is lower than the threshold, otherwise, the second and more detailed feedback is returned to the student. Feedback could be a simple message or a link to a chapter, a document, a web page, an exercise, etc. The experience of the teacher in programming teaching and programs assessing plays an important role when predicting the errors and the plausible causes for each exercise to define the corresponding corrective feedback. Although the error base in CLAAS includes the most common errors made by novice students in C programming, it is continuously expanding through the emergence of new errors added by evaluators throughout the use of CLAAS.

Example1: compiler message in the case of misspelling error



```
C:\>clang progc\example1.c
progc\example1.c:3:2: warning: implicit declaration of function 'prinft' is invalid in C99 [-Wimplicit-function-declaration]
        prinft("hello C word!!");
        ^
```

The compiler used in CLAAS is Clang, an open source C and C-like language compiler. The reports returned by Clang are aimed to be detailed, specific and expressive, as well as machine-readable [20]. In example 1, according to the compiler, the problem is the invalid declaration of a function. For a novice student in programming, this message is ambiguous and does not explain the error and the necessary fixes. CLAAS reports a similarity between the token 'prinft' in the compiler message and a function in C language and identifies the error within the misspelling errors category in the base. Here, the first feedback may consist of the name of the error's category while the second one may warn the student that the word "prinft" is similar to a function in C language or redirect the student to a section in the course or exercises related to C programming basics as shown in Figure 3.



Écrire un programme qui demande à l'utilisateur de saisir deux entiers A et B et l'informe ensuite si leur produit est négatif ou positif.
Le programme doit afficher « PRODUIT NEGATIF » ou « PRODUIT POSITIF » ou « PRODUIT NUL ».

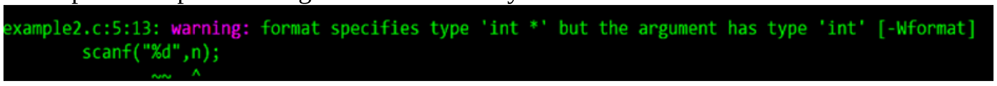
```c
#include <stdio.h>
int main(){
    prinft("hello C word!!");
    return 0;
}
```

```
c:\code\example1.c:5:8: warning: implicit declaration of function 'prinft' is invalid in C99
      [-Wimplicit-function-declaration]
       prinft("hello C word!!");
       ^
1 warning generated.
```

Erreur d'orthographe !

Figure 3. A feedback example in CLAAS for a compiling error

Example2: compiler message in the case of a symbol error



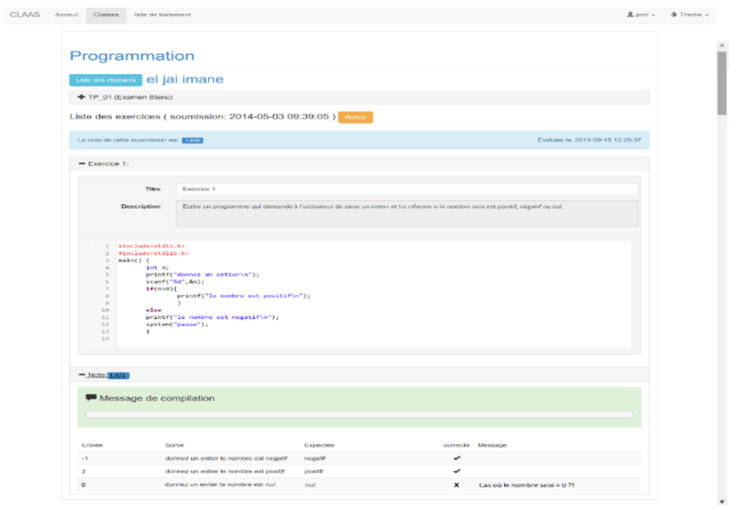
```
example2.c:5:13: warning: format specifies type 'int *' but the argument has type 'int' [-Wformat]
        scanf("%d",n);
               ~~   ^
```

In this example, there was an omission of the symbol & in the scanf function. This error is not detecTable as a compiling error, but the execution of the program does not produce the expected outputs. Nevertheless, unlike several other compilers, Clang points this as a warning which flags an incompatibility between the used variable's type and the argument used in scanf, an ambiguous message for the learner.

Based on this warning and on the scanf signature in C language, CLAAS performs a parsing into the erroneous portion of the code to locate the error and its corresponding category in the errors base. In this example, as a first feed-back, the teacher could remind the student that each argument in scanf function must be a memory address where the converted result is written. In the second more detailed feed-back, he could indicate that & symbol should precede each variable in scanf in order to modify his content.

2.1.2. Feed-back in the case of logical errors

Throughout dynamic program analysis, when the evaluated program succeeds the compilation stage, it is then assessed using test-cases. A test-case failure may be due to a logical error in the program, which occurs when the exercise specification is not respected. For example, when writing a C program to check whether a number is positive or negative or zero, most novices omit to consider the case of input = 0. This causes the failure of the corresponding test-case. Within the exercise creation, the teacher assigns a feedback to each test-case according to the possible cause of the test-case failure. Figure 4 presents a proposed feedback which draws the attention of the student to the missed part in his program.



Programmation

el jai imane

TP_01 (Examen Blanc)

Liste des exercices (soumission: 2014-05-03 09:39:05)

La note de cette soumission est: [illegible] Evaluée le: 2014-09-15 12:29:37

Exercice 1:

Titre: Exercice 1

Description: Ecrire un programme qui demande à l'utilisateur de saisir un entier et lui informe si le nombre saisi est positif, négatif ou nul.

```c
#include<stdio.h>
#include<stdlib.h>
main() {
        int n;
        printf("donnez un entier\n");
        scanf("%d",&n);
        if(n>0){
                printf("le nombre est positif\n");
                }
        else
        printf("le nombre est negatif\n");
        system("pause");
        }
```

Note: 1.5/2

Entrée	Sortie	Expectée	correcte	Message
-1	donnez un entier le nombre est negatif	negatif	✓	
2	donnez un entier le nombre est positif	positif	✓	
0	donnez un entier le nombre est nul	nul	✗	Cas où le nombre saisi = 0 ?!

Figure 1. A feedback example after a test case failure in CLAAS

2.2. Feedback through static analysis of programs in CLAAS

Static analysis of programs refers to the process of examining code without executing it to capture the defects [21, 22]. Similarity analysis is a static method which consists of measuring similarity between two programs. This method is adopted in CLAAS to assess submitted programs and provide an elaborate feed-back for the student as shown in Figure 5.

Static program analysis based on semantic similarity measurement is used in CLAAS in order to grade the student program according to the degree of its similarity to a model program [23]. Thus, the compared programs are sliced into a set of blocs, then each bloc in the student program is sequentially compared to all the blocs in the model program.The performed comparisons are based on symbolic execution outputs of the blocs in the programs. These comparisons generate semantic similarity rates, used in a weighted formula to calculate the grade of the evaluated program. The guiding principle behind semantic similarity measurement is that two structurally different programs could be semantically equivalent. The grading process was discussed in a previous work [16]. Semantic similarity measurement is used to assist the student in identifying the errors in three different cases.

a) Case n°1: The blocs having a low similarity degree are mainly suspicious. Therefore, they are flagged to be checked by the students for errors detection.
b) Case n°2: Semantic similarity measurement serves as well to detect dead code in the student program when the evaluated program has a full mark, and a bloc showed no similarity to none of the blocs in the model program through all the performed comparisons. This bloc is then identified as unnecessary, and the returned feed-back suggests its removal from the student program. Such feed-back assists the student in acquiring good habits for writing optimized code.
c) Case n°3: Through assessing large numbers of assignments, we noticed that students frequently submit incomplete programs. Using semantic similarity measurement, we can detect incomplete programs, when a bloc in the model program has no similar bloc in the student program. In this case, the feed-back informs the students that his program needs to be completed and returns the feed-back associated by the evaluator to the corresponding bloc in the model program that should be added to complete the student program.

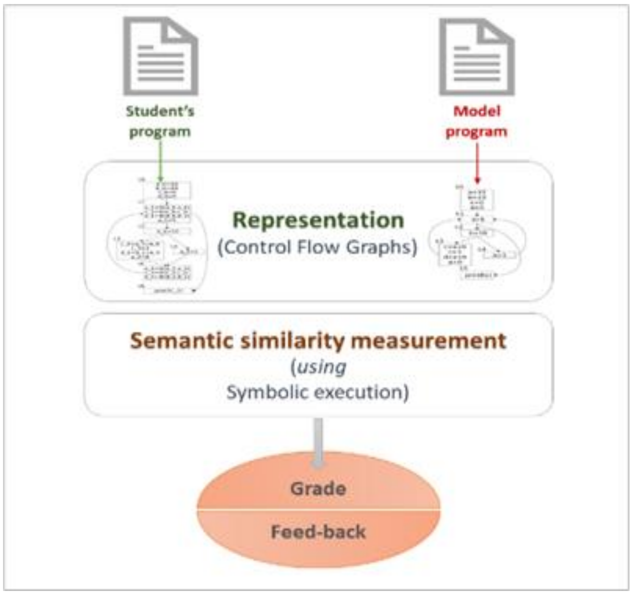
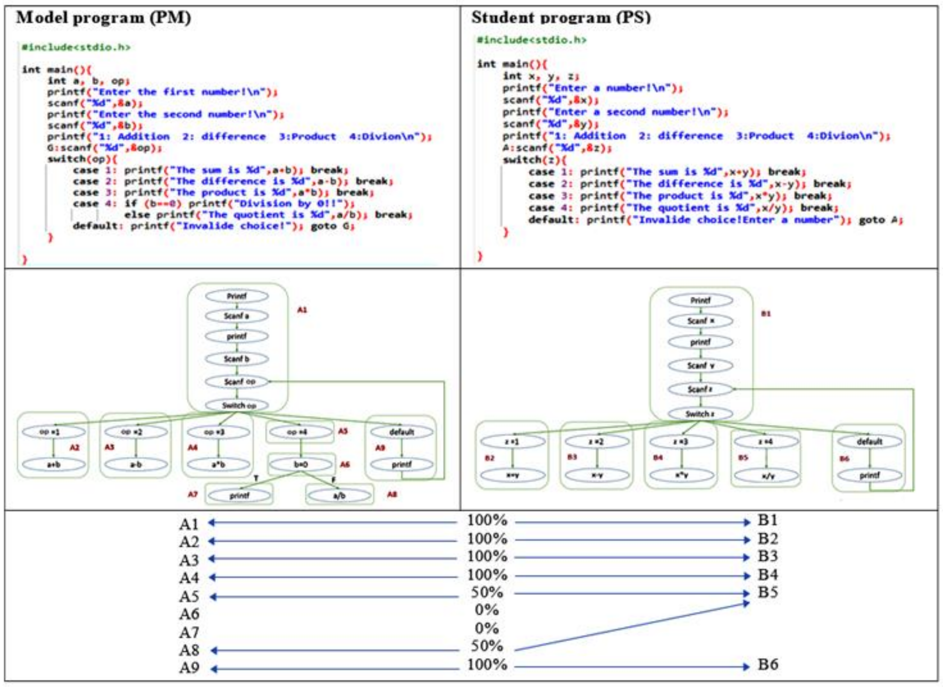


Figure 5. Static analysis method for program assessment in CLAAS

Let's consider a programming problem which consists of writing a program to calculate the sum, product, quotient or the difference of two entered numbers, according to the chosen operation. Figure 6 presents a model program (PM), a student program (PS) and semantic similarity percentages between the blocks in PM and PS. In CLAAS, the evaluator assigns a feed-back to the blocs of the model programs. The bloc A6 in the model program have no similar bloc in the evaluated program. This bloc handles the case of a division by zero. In this case, the feed-back returned to the student consists of reporting that the bloc covering the division by zero is missed in his program. It could also return the instructions in the bloc A6 according to the evaluator's setting.



Model program (PM)

```c
#include<stdio.h>

int main(){
    int a, b, op;
    printf("Enter the first number!\n");
    scanf("%d",&a);
    printf("Enter the second number!\n");
    scanf("%d",&b);
    printf("1: Addition  2: difference  3:Product  4:Divion\n");
    G:scanf("%d",&op);
    switch(op){
        case 1: printf("The sum is %d",a+b); break;
        case 2: printf("The difference is %d",a-b); break;
        case 3: printf("The product is %d",a*b); break;
        case 4: if (b==0) printf("Division by 0!!");
                else printf("The quotient is %d",a/b); break;
        default: printf("Invalide choice!"); goto G;
    }

}
```

Student program (PS)

```c
#include<stdio.h>

int main(){
    int x, y, z;
    printf("Enter a number!\n");
    scanf("%d",&x);
    printf("Enter a second number!\n");
    scanf("%d",&y);
    printf("1: Addition  2: difference  3:Product  4:Divion\n");
    A:scanf("%d",&z);
    switch(z){
        case 1: printf("The sum is %d",x+y); break;
        case 2: printf("The difference is %d",x-y); break;
        case 3: printf("The product is %d",x*y); break;
        case 4: printf("The quotient is %d",x/y); break;
        default: printf("Invalide choice!Enter a number"); goto A;
    }

}
```

A1	100%	B1
A2	100%	B2
A3	100%	B3
A4	100%	B4
A5	50%	B5
A6	0%	
A7	0%	
A8	50%	B5
A9	100%	B6

Figure 6. Control flow graph and similarity rates between a model and a student program

2.3. Compiling errors auto-repair

Compiling errors are frequently committed by students. They are the easiest to fix, however, students face difficulties and spend a long time to detect, localize and fix this kind of errors [24, 25]. Most students submit uncompilable programs while based on correct algorithms. Automated repairing of errors consists first of their detection and localizing. It is activated after several attempts made by the student to submit a compiled program. The number of authorized resubmissions (n) is defined by the teacher among the exercise parameters as shown in Figure 7.

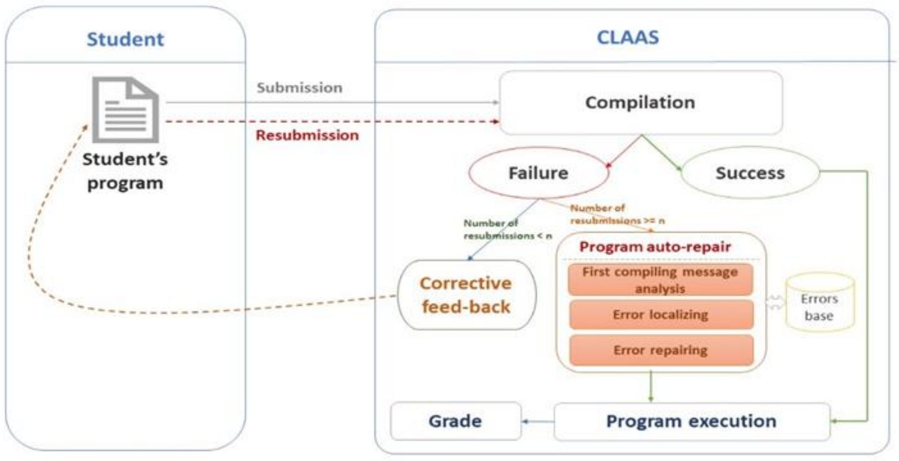


Figure 7. Program auto-repair process in the case of a compiling error

The compiler is still a strong tool for errors detection. Thus, the errors repairing is performed by the correction unit in CLAAS, using the compiler logs to perform the necessary fixes (Figure 7). The correction unit presented above, analyses the compiling reports and considers the first reported error. To localize the error in the source code, the correction unit performs syntactical parsing of the erroneous portion in the code, based on the number of the line contained in the compiler message. The automated repairing in CLAAS is performed through an iterative process. Hence, after repairing the first message in the compiler log, the correction unit then triggers a new compilation. The process is interrupted when it exceeds the max number of iterations is reached or when the correction unit is not able to correct the remaining errors. Examples of automated repairing of compiling errors

a) Example 1: exemple1.c:3:23: error: expected ';' after expression
The missed semicolon is a frequently committed error in novices' programs. This error is well captured and localized in the compiler message. In this case, the correction unit uses this information to insert the missed character and generates a new code.

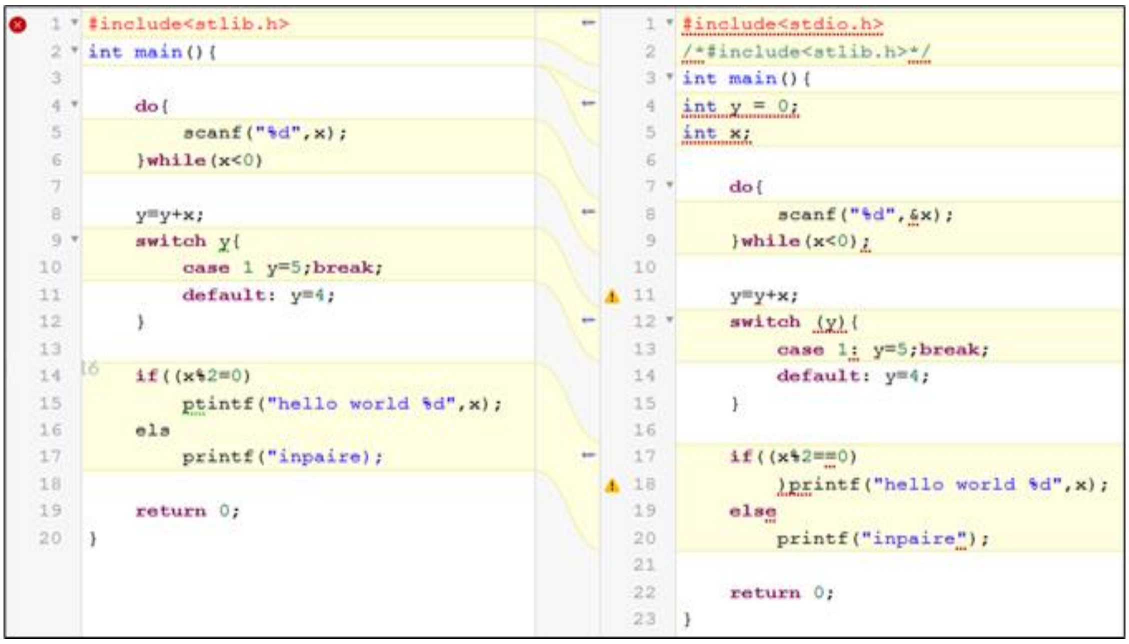
b) Example 2: exemple2.c:5:13: warning: format specifies type 'int *' but the agrument has type 'int'[-*Wformat]*
The missed & in scanf function is another frequent error in programming. This error causes no compilation failure; however, execution outputs are invalid. Clang returns a warning in this case to signal an incompatibility between the used variable and the argument in scanf function. In this case, the correction unit performs a parsing of the scanf function signature in the erroneous line in the code and the missed & is then inserted to generate a new code for the next compilation.

c) Example 3: exemple3.c:6:17: error: expected ';' in 'for' statement specifier

for(i=0,i<a;i++){

In this example, the error is detected by the compiler and localized before the closing parenthesis in the loop. Inserting the missing character in the location returned by the compiler resolves the compiling error; however, it generates invalid execution outputs. In this case, the correction unit analyses the erroneous line in the code, to detect the violation in the "for" loop syntax and inserts the missed semicolon in the appropriate place. In such cases, the correction unit uses only the compiler information about the erroneous line, and performs the appropriate correction based on the 'for' function signature.

Figure 8 presents a comparative display of a program with 15 different compiling errors repaired after 6 iterations by the correction unit in CLAAS.



```
#include<stlib.h>
int main(){

    do{
        scanf("%d",x);
    }while(x<0)

    y=y+x;
    switch y{
        case 1 y=5;break;
        default: y=4;
    }

    if((x%2=0)
        ptintf("hello world %d",x);
    els
        printf("inpaire);

    return 0;
}
```

```
#include<stdio.h>
/*#include<stlib.h>*/
int main(){
int y = 0;
int x;

    do{
        scanf("%d",&x);
    }while(x<0);

    y=y+x;
    switch (y){
        case 1: y=5;break;
        default: y=4;
    }

    if((x%2==0)
        )printf("hello world %d",x);
    else
        printf("inpaire");

    return 0;
}
```

Figure 8. Errors mapping to the performed corrections

3. EXPERIMENTATION

In order to check the accuracy of compiling errors auto-repair process in CLAAS, 185 real programs were submitted by students in first-year of bachelor's degree of Science & Technics to resolve 5 programming exercises within a mock exam. The following graphs presents the results of automated assessment before and after applying the automated repairing of compiling errors, compared to manual assessment as shown in Figure 9-13.

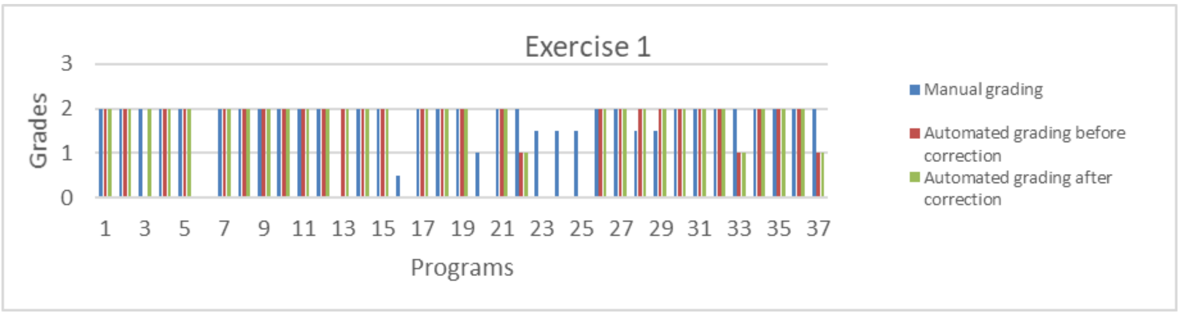


Figure 2. Exercice 1: C program checking whether the input integer number is positive or negative or zero

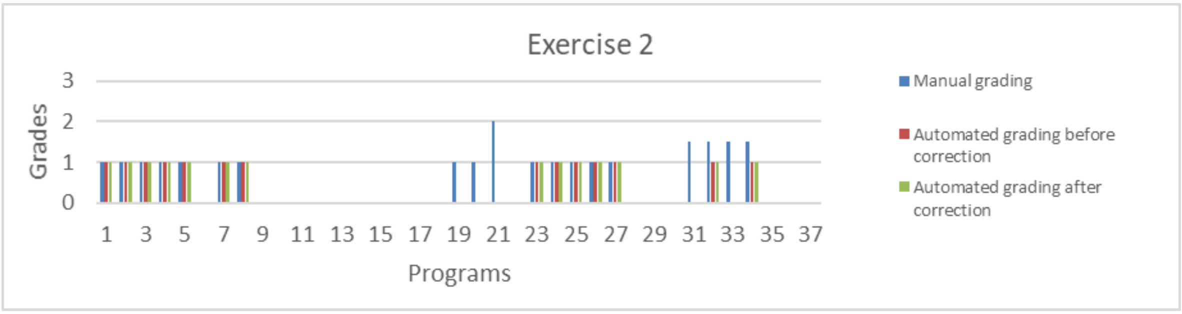


Figure 10. Exercice 2: C program which asks to input 5 grades between 0 and 20 and counts the grades >=10

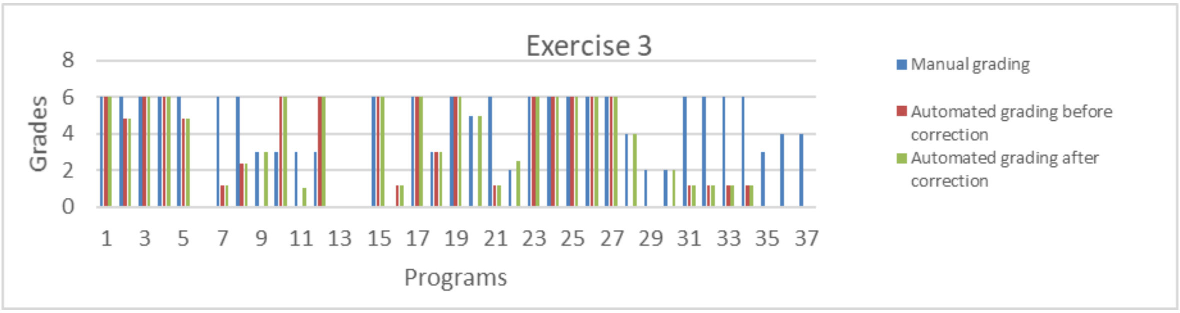


Figure 11**.** Exercise 3: C program to perform addition, subtraction, multiplication and division according to the chosen operation

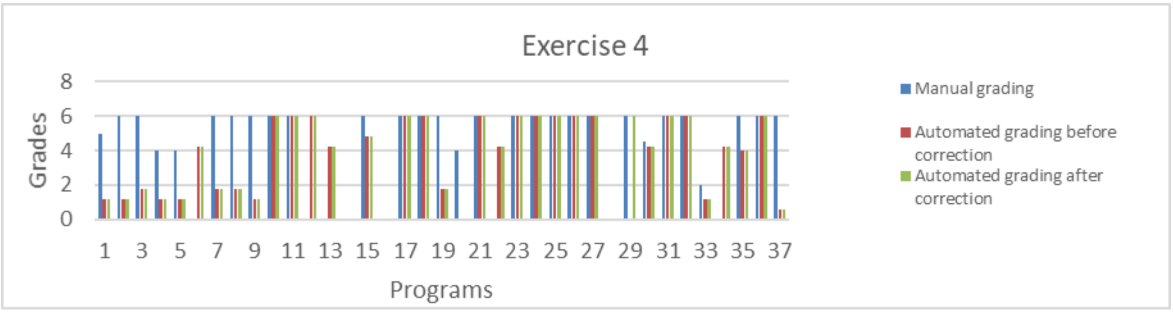


Figure 12. Exercise 4: C program to print stars according to the input integer

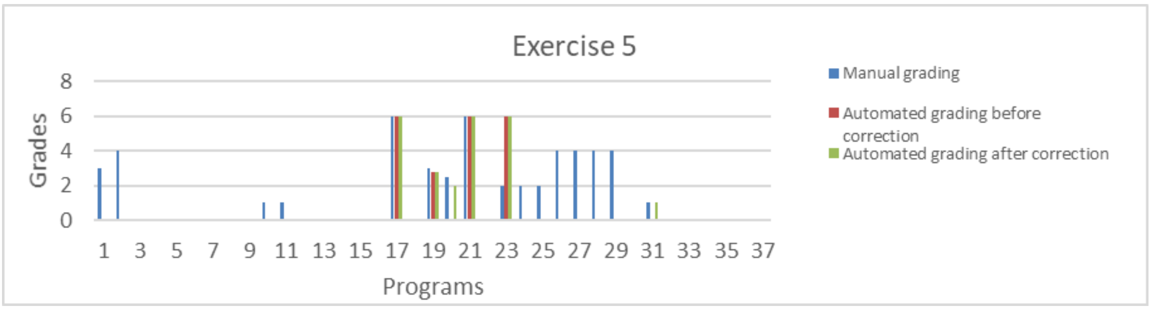


Figure 13. Exercise 5: C program which asks to input an integer <=100 while the input number is not valid

The results show that 20% of the evaluated programs have a null grade. For 55% of these programs, compilation failure interrupted the assessment process; consequently, they did not reach test-cases analysis. Within this experiment, 28 errors from different categories were repaired by the correction unit in CLAAS. Table 2 presents the averages of grades obtained from manual and automated grading, before and after implementing program auto-repair function. In addition, we calculated the grade precision, a degree between 0% and 100% which reflects the approximation between the manual grading results and the automated ones before and after applying the automated repairing of compiling errors. The grade precision is calculated using the formula (1).

$$GP = \left(1 - \frac{|note\ manuelle - note\ du\ système|}{note\ maximale}\right) \times 100 \quad (1)$$

Table 2. Averages of manual and automated grading

	Marking scheme	Average grades of manual grading	Grades of automated grading		Grades of automated grading using program auto-repair	
			Average	Grade precision	Average	Grade precision
Exercise1	2/20	1,76	1,54	89%	1,59	92%
Exercise2	2/20	0,59	0,38	89%	0,38	89%
Exercise3	6/20	4,35	2,74	73%	3,21	81%
Exercise4	6/20	4,36	3,48	85%	3,64	88%
Exercise5	4/20	1,34	0,56	81%	0,64	83%
Total grade	20/20	12,41	8,70	81%	9,47	85%

4. DISCUSSION

The various categories of feed-back proposed within this work meet several criterions of valuable feed-back, which should be informative, timely, consistent, clearly communicated and useful for students. CLAAS provides feedbacks with different amount of elaboration, delivered progressively according to the performance of the student. The feed-back generation process in CLAAS was designed to meet the human proceeding in assisting students within programming courses. For this purpose, both static and dynamic methods were used to consider different aspects of the evaluated program and provide useful and continuous feed-back, from flagging the erroneous or missing lines in the code, to reporting the origin of a test-case failure or a compiling problem.

Despite the various feed-back generated by CLAAS, the student could still submit a program with compiling errors that cannot be executed and assessed whilst it might be partially correct. This prevents the assessment of the program correctness and thus the algorithm on which the solution is based. Program auto-repair is implemented in CLAAS within the correction unit, which localizes and fixes compiling errors in the student program. Thus, the student receives feed-back during all the assessment process, from compiling to test-cases execution. During the experiments within practical sessions, we noticed that students were asking less for teachers' assistance and were motivated to use the delivered feed-back to solve the encountered problems.

5. CONCLUSION

This work is the continuation of previous research in the field of automated assessment of C programs. It focuses on feed-back as a paramount factor which assists the student through the learning process. The use of static and dynamic program analysis, in addition to the automated repair of programs provides a variety of valuable feed-backs for the student. This is ensured through considering different aspects of the evaluated program to assist the student at various levels within the evaluation process.

By the means of the feed-back integrated in CLAAS, the student can face difficulties due to the lack of compiling errors understanding, he can also investigate the cause of the program failure to produce the expected behavior, as well as appreciating the semantic similarity between his program and a program proposed by the teacher to learn good programming habits and submit a complete, functional and correct program. The integration of such kinds of feed-back can serve to envision a new generation of compilers dedicated to the assessment of students' programs in the context of programming learning, which considers specific needs and requirements of students in this field. As future work, we are planning a long-term performance review among the use of feed-back in CLAAS, since measuring the impact of feed-back integration on students' learning takes several years and a rigorous process to abolish subjectivity due to the context, human factors, etc. We also aim at covering additional types of errors to expand the usefulness of CLAAS.

REFERENCES

[1] I. Kožuh, R. Krajnc, L. J. Hadjileontiadis, and M. Debevc, "Assessment of problem solving ability in novice programmers," *PLoS ONE*, vol. 13, no. 9, pp. 1–21, 2018.

[2] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 14–18, Sep. 2005.

[3] A. T. Corbett and J. R. Anderson, "Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes," in *ACM Conference on Human Factors in Computing Systems*, 2001, pp. 245–252.

[4] J. Hattie and H. Timperley, "The Power of Feedback," *Review of Educational Research*, vol. 77, no. 1, pp. 81–112, 2007.

[5] R. Singh *et al.*, "Automated Feedback Generation for Introductory Programming Assignments," *ACM SIGPLAN Notices*, pp. 15–26, 2013.

[6] J. C. Caiza, J. M. Del Alamo, and J. M. Del Álamo Ramiro, "Programming Assignments Automatic Grading: Review of Tools and Implementations," *7th International Technology, Education and Development Conference (INTED2013)*, pp. 5691–5700, 2013.

[7] Y. Dong, T. W. Price, and T. Barnes, "Generating Data-driven Hints for Open-ended Programming," *Proceedings of the 9th International Conference on Educational Data Mining*, pp. 191–198, 2016.

[8] B. Jeffries, T. Baldwin, M. Zalk, and B. Taylor, "Online tutoring to support programming exercises," *ACE 2020 - Proceedings of the 22nd Australasian Computing Education Conference, Held in conjunction with Australasian Computer Science Week*, pp. 56–65, 2020.

[9] O. Mirmotahari, Y. Berg, S. Gjessing, E. Fremstad, and C. Damsa, "A case-study of automated feedback assessment," in *IEEE Global Engineering Education Conference, EDUCON*, 2019, vol. April-2019, pp. 1190–1197.

[10] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Transactions on Computing Education*, vol. 19, no. 1, Sep. 2018.

[11] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: what can help novices?," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2016.

[12] B. A. Becker, "An effective approach to enhancing compiler error messages," *SIGCSE 2016 - Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 126–131, 2016.

[13] P. Denny, A. Luxton-reilly, and D. Carpenter, "Enhancing Syntax Error Messages Appears Ineffectual," in *Proceedings of the 2014 conference on Innovation & technology in computer science education.*, 2014, pp. 273–278.

[14] C. Watson, F. W. B. Li, and R. W. H. Lau, "Learning Programming Languages through Corrective Feedback and Concept Visualisation," in *International Conference on Web-Based Learning*, 2011, pp. 11–20.

[15] S. Mernissi Arifi, I. Nait Abdallah Ouali, A. Zahi, and R. Benabbou, "Automatic program assessment using static and dynamic analysis," in *Proceedings of 2015 IEEE World Conference on Complex Systems, WCCS 2015*, 2015.

[16] S. Mernissi Arifi, A. Zahi, and R. Benabbou, "Semantic similarity based evaluation for C programs through the use of symbolic execution," in *IEEE Global Engineering Education Conference, EDUCON*, 2016, vol. 10-13-Apri.

[17] S. Mernissi Arifi, I. Nait Abdallah Ouali, R. Benabbou, and A. Zahi, "Automated fault localizing and correction in dynamically analyzed programs," in *Colloquium in Information Science and Technology, CIST*, 2016.

[18] J. Munson and E. Schilling, "Analyzing novice programmers' response to compiler error messages," *Journal of Computing Sciences in Colleges*, vol. 31, no. 3, pp. 53–61, 2016.

[19] R. Pettit, J. Homer, and R. Gee, "Do enhanced compiler error messages help students? Results inconclusive," *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, pp. 465–470, 2017.

[20] C. Guntli, "Architecture of clang," *University of Applied Science in Rapperswil*, pp. 1–12, 2011.

[21] K. M. Ala-mutka, "A Survey of Automated Assessment Approaches for Programming Assignments," *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005.

[22] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5308 LNCS, pp. 238–255, 2008.

[23] S. Mernissi Arifi, R. Benabbou, and A. Zahi, "A New Similarity-based Method for Assessing Programming Assignments using Symbolic Execution," *International Journal of Applied Engineering Research*, vol. 13, no. 4, pp. 1963–1981, 2018.

[24] S. Parihar, R. Das, Z. Dadachanji, A. Karkare, P. K. Singh, and A. Bhattacharya, "Automatic grading and feedback using program repair for introductory programming courses," *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, vol. Part F128680, pp. 92–97, 2017.

[25] C. Watson, F. W. B. Li, and J. L. Godwin, "BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair," *International Conference on Web-Based Learning*, no. September, pp. 228–239, 2012.

BIOGRAPHIES OF AUTHORS

Sara Mernissi Arifi got his Ph.D. in Computer Science from Sidi Mohamed Ben Abdellah University in Fez, Morocco. Currently acting as a teacher of computer science in a secondary school since 2005. His main interests are in automated assessment in elearning environments, program analysis and web development.

Rachid Ben Abbou is a Professor in Computer Science Department at Sidi Mohamed Ben Abbdellah University in Fez, Morocco since 1997. Member of Laboratory of Intelligent Systems and Applications (LSIA). His research interests concern ad hoc network, VANET, security in Cloud Computing, e-learning, Automatic Assessment.

Azeddine Zahi is a full Professor of computer science at the Faculty of Sciences and Technology of Fez University in Morocco, since 1995. He received his Master and Doctorate of 3rd Cycle in Computer Science from the University of Mohamed V in 1994 and 1997 respectively. He is an active member of the Intelligent Systems and Applications Laboratory (LSIA). His research area is related to Data Mining and Artificial Intelligence. He is concerned with the use of data mining process, machine learning methods and artificial intelligence techniques in the fields such as software project cost estimation, pattern recognition, Adhoc Networks and automatic evaluation of computer programs. He is also interested by new trends in Data mining and data analytics that aimed to deal with many issues arising in complex environment, such as uncertainty, incompleteness, heterogeneity and Bigness.