Automatic Analysis and Evaluation of Student Source Codes

Ádám Pintér*, Sándor Szénási†‡

* Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University, pinter.adam@nik.uni-obuda.hu
† John von Neumann Faculty of Informatics, Óbuda University, szenasi.sandor@nik.uni-obuda.hu
‡ Faculty of Economics, J. Selye University, szenasis@ujs.sk

***Abstract*—An essential part of education is the evaluation of students and the facilitation of students' studies through feedback. During evaluation and feedback, it is important to ensure that it applies equally to all students, is fair and objective. For simple (e.g., multiple choice) tests, their evaluation can be easily automated; however, source code evaluation is still done manually and often incorrectly.**

In our article, we present the model we used evaluating solutions for OE-NIK SzTF2 student homework written in C#. The program based on this model is able to both syntactically examine the internal structure of the code and to run the code and compare the input-output pairs. In addition, different metrics are defined for each source code examined, based on which instructors can make additional suggestions to students to improve their programming skills. The consequence is that with the help of the proposed system it is possible not only to decide whether a code is correct or not, but also due to a deeper analysis, further curious details are revealed.

***Index Terms*—analysis, assessment, source code, C#**

I. Introduction

Evaluation is an essential part of education and is useful both for students who receive feedback on the progress of their studies and for educators who can assess whether or not students have achieved their goal. In this approach, it has been shown that most students choose the easier path to achieve the best result during evaluation, instead of acquiring comprehensive knowledge, which would also benefit for companies. [1] It follows that continuous assessment of the course can be used to guide and improve students' learning process. As manual evaluation requires a significant effort even in the case of small groups, it is advisable to try to find ways to automate some or all of the work. Automation is further justified by the fact that it is almost impossible to take all aspects into account in the evaluation, and it is also very rare for two instructors to perform the assessment based on the same criteria and arrive at the same result. Of course, the grades developed in this way will depend heavily on the instructor conducting the assessment, which may unfairly affect many students.

A. Manual and automatic evaluation

When testing source code manually, it is usually not possible for the instructor to evaluate the solutions in all respects. Many times he or she has only time to perform the most basic features, formal requirements, and minimal runtime checks.

The automatic evaluation was run on the homework of the students in the second semester C# programming subject in OE-NIK, which includes inheritance, abstract classes, interfaces, concatenated lists, binary trees, graphs, and hash tables.

Students always had to implement only one class when solving the homework, which either required the implementation of an interface or had to be a descendant of an abstract class. If this had to be evaluated by the instructors, each submitted student solution (class) would have had to be manually copied into the sample project attached to the task, translated, and then checked for the correct operation of the solution, which may include several error possibilities.

B. The advantage of automatic evaluation

Poor evaluation is disadvantageous to students and feedback may be misinterpreted. In contrast to manual evaluation, automatic evaluation has the following advantageous features:

- *Fast*. Machine evaluation takes place almost immediately, and the student receives feedback on the result and details of any errors after the completion of the examination.
- *Fair*. Assessment is consistent, identical errors are evaluated in the same way, without instructor influence.
- *Independent*. The evaluation of a given task is not affected by the student's performance in the previous exam or other prejudices related to the student.

Unfortunately, the implementation of a perfect evaluation system is far from reality, the primary reason being that a solution to a problem can occur in almost infinite ways. To handle this diversity, the programming language must be able to do introspection (to examine its own structure and state at runtime) and intercession (to change its structure / state / operation at runtime). However, most languages do not have these capabilities, or if they are available, it is only in a limited form. In this project, we go beyond these limitations and present a system that is able to fully evaluate the solutions of (sub)problems.

C. The task to be solved

We designed an application that can evaluate various programming tasks. During the evaluation, we have a student solution and a sample project (template) that contains what is needed to interpret the student solution. During the study,

978-1-7281-8340-4/20/$31.00 ©2020 IEEE

the template and the student solution are interpreted and run together, and finally an evaluation report is created.

The rest of this paper is organized as follows. In the next Section, we present the related work of automatic assessments of source code. Then, in Section 3, we present our model and the functionality of a custom automatic assessments system, and discuss its limitations. Experiments of our implementations that are being used in real university courses are presented in Section 4. And finally, we present our conclusion in the last section.

II. Related Work

Pears et al. [2] grouped the teaching of programming tools as follows:

- Programming support tools. Tools that improve the programming experience help programmers understand how programs work. Examples include programming environments, debuggers, and profiling (i.e.: BlueJ [3]).
- Microworlds. Systems specifically designed to teach programming and include tools and interfaces to learn their various concepts (i.e.: Karel [4], Alice [5]).
- Visualization tools. [6] Allows step-by-step visualization of algorithms for easier understanding and debugging (i.e.: Animal [7], Jeliot [8]).
- Automatic Assessments (AA) tools. Tools to help educators in the program evaluation process (i.e.: TRAKLA2 [9], WEBCAT [10], BOSS [11]).

In our article, we focus on the field of AA tools, including those that deal with source codes. Most of these devices use black-box technique, i.e. they evaluate the student code by comparing the outputs given to the inputs. [12]–[18] These tools first compile the student's source, run all or part of it (function call) with a predefined set of inputs, and then compare it to the expected outputs. The biggest disadvantage of this method is that the student's code will not be analyzed, or only minimally, and its operation will be based only on the output (in some cases, runtime and memory usage are usually examined). More advanced systems using such a technique use random test cases or some pre-constructed structure (usually the correct algorithm) instead of pre-written tests to construct the results. [19]–[21] There are also language-independent black-box systems, which are characterized by the fact that, in addition to the code to be tested, the corresponding compiler and test cases are received as one parameter.

The number of systems using the white box approach is significantly less than that of black box systems. When using this technique, a more in-depth analysis of the code is also performed, typically with the help of a predetermined correct solution provided by an instructor. [18], [22]

Our approach is to create a hybrid system that, in addition to comparing the output to the input, also performs syntactic analysis of the source code and is able to search for duplications in the source code.

III. Methodology

In this undertaking, we have created a model of a hybrid solution (Figure 1) for source code verification that is able to syntactically, semantically and operationally verify student solutions. The model includes the following main steps:

- Download the list of students in the course from the Moodle system, based on which the presence or absence of the students' homework can be checked.
- Collect the templates and student solutions for the tasks from the Moodle system, then create a skeleton file that contains the templates and the unit tests needed to check the tasks at runtime, as well as the syntax rules for the task.
- Transforming students' source code into an abstract syntax tree (AST), pre-processing the solution based on AST, and deeper syntactic analysis.
- Transform the skeleton file into AST and then merge it with the student solutions to get the student standalone solution for a given homework.
- Compile standalone source code and run unit tests.
- Preparation of a detailed html report and a summary report in excel per student.

A. Requirements for usage of the Model

For the model to work properly, several aspects must be met for the programming language in which the tasks are written. The need for these is justified by the fact that in the case of low-level languages, it is difficult to create a library that provides the following functionalities:

- *Source code analysis and manipulation.* The model needs not only to run the student solution, but also to analyze and modify it, so for example, it must be able to calculate the number of lines of code or modify the contents of the namespace. For this, it is advisable to be able to transform the source code into AST in such a way that all the small details of the code are preserved. An additional requirement for a tree is to be able to crawl, analyze, and modify it in such a way that the tree retains its semantic correctness.
- *Runtime manipulation of source code.* Reflection is a commonly used technique that provides the ability to modify source code already loaded into memory at runtime or to examine certain states. This is a relatively advanced technique that is only recommended in exceptional cases, as a poorly implemented reflexive operation can lead to unexpected behavior in the source code and pose a security risk in addition to performance issues.

The model is primarily used to evaluate homework written in C#. The language of choice has advanced reflection capability, however, the Roslyn API [23] was used to analyze and manipulate the source code, which was developed for compiling and analyzing the code and forms the basis of the .Net Core.

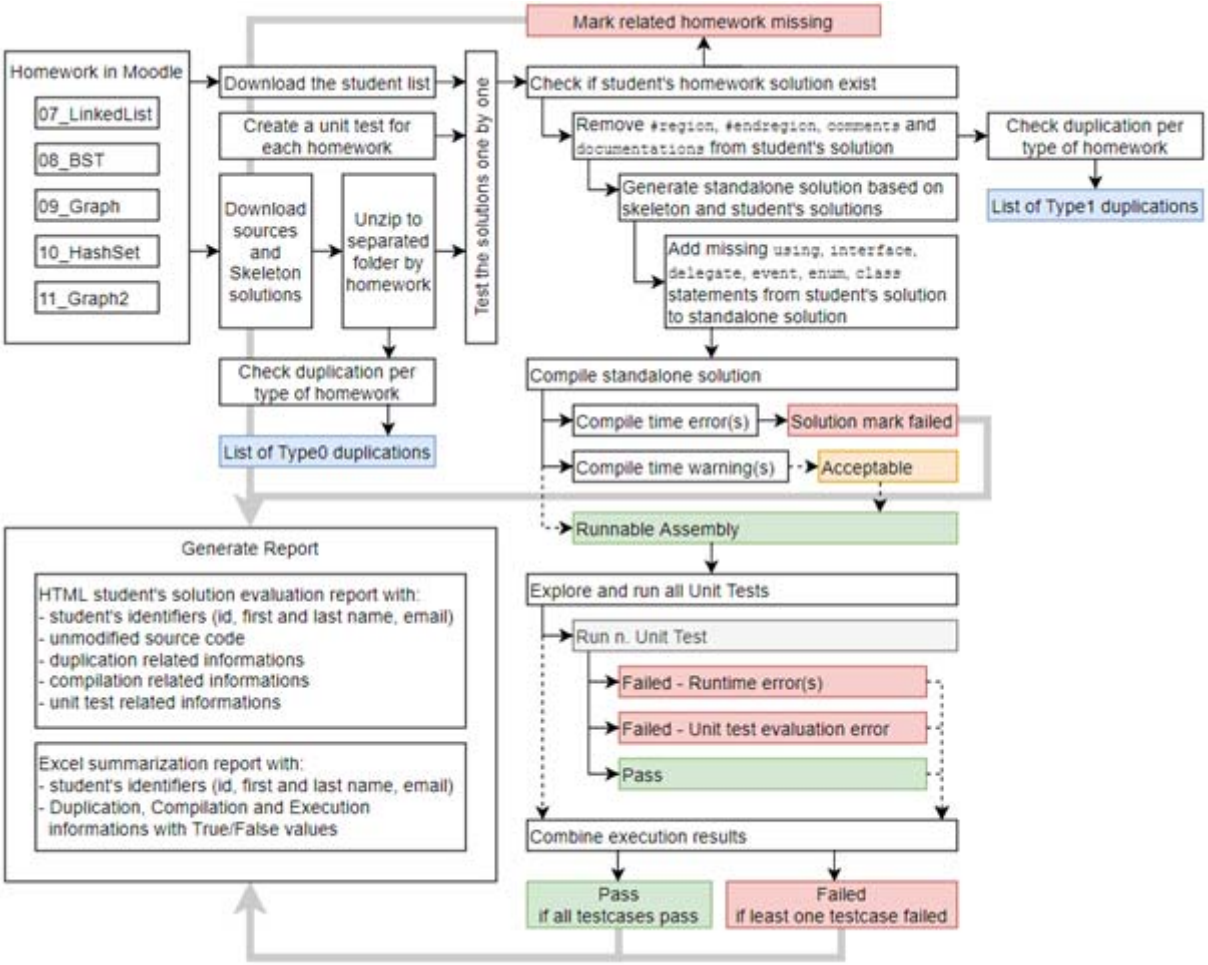

Fig. 1. A model for evaluating student homework that includes exit lines from the model with (thick gray) lines.

B. Student Dataset and Templates

The dataset used consists of two parts: the student solution and the template. In each case, students solve a task that includes a template. The template project contains either the ancestor(s) of the class to be created or some interface(s) to be used, or an empty project in case the task can be created more freely. Once the student has completed the homework, all they need to do is to submit the single file that contains their own implementation. There is no need to upload the template project; it is the same for all solutions.

During data collection (beyond the list of students in the course), the template project and all student solutions are downloaded for each homework. Once downloaded, a skeleton file should be created based on the template, which combines the contents of the project into a single file for easier management, and is supplemented with unit tests for the task.

C. Preprocessing the Source Code Files

After collecting and preparing the necessary files, the students' solutions are pre-processed and combined with the skeleton file, resulting in a standalone solution. During preprocessing, duplications are also examined at two different locations, with the aim of filtering out and marking identical homeworks. The preprocessing is performed as follows:

- Search for character-level (Level 0) duplicates that assign solutions to each student's solution that match it. The existence of duplications does not affect the processing process, in such cases it is the responsibility of the instructor supervising the evaluation to make the further decision.
- Transform the student solution to AST and then do a minimal cleanup: remove comments, documentation, #region and #endregion directives, and replace indents with spaces. The refined source code will be followed by a (Level 1) duplication search, the results of which will also be evaluated by the instructor.
- Transform the skeleton file into AST and then merge it with the student's solution so that only those parts of the student's tree that are not in the skeleton are highlighted. This constraint ensures that the student implements things in the form required by the assignment.
- The last step is to create a standalone solution from the merged AST, which is a text representation of source code formatted with spaces.

D. Compile and Verify

After the standalone solution is prepared, it is compiled. During the process, if the compiler encounters a compile time error, the evaluation of the solution ends, and the solution contains a syntax error. Upon successful compilation (which may include CompileTimeWarning), an in-memory application is generated that is syntactically correct and executable. Unit tests are run on this memory stream if the function implemented by the student can be successfully extracted from it. Otherwise, the solution is marked as incorrect.

E. Syntactic Check

Students' unmodified source code, converted to AST, also passes through a parser based on rule-based verification of declarations in the code. The instructor can set up different rules during syntactic checking, such as requiring method calls in some part of the code, or even imposing constraints on naming conventions that you can check for in this way. The outcome of each rule can be successful or unsuccessful. Rules can be defined in XML format, where a rule is structured as follows:

- *nodeType*: the type of nodes relevant to the rule. When specifying the type, it is possible to define a longer (but not contiguous, omitting intermediate elements) route, separated by dots. Describing the route is important in cases where you want to specify the item to be examined in the tree. The Roslyn defined node name should be used to determine the type.
- *nodeKeywords*: the name(s) of the node(s) to which the rule applies. The ability to specify multiple values provides the ability to find complex structures. There is an AND relationship between the listed values.
- *constraints*: the rule(s) that either requires the structure or prohibits its presence in the code. Its values can be: NotAllowed, Mandatory, Minimum X, Maximum X, Exactly X.
- *failMsg*: the error message associated with the rule, which is displayed to the student if the rule is met.

F. Functionality Check with Unit Tests

During the preparation of the unit tests, we used the NUnit framework [24], in which the well-known functionalities of the unit tests are readily available. When preparing test cases, it may be difficult to find and instantiate the class to be tested. Namely, in relation to student solutions tied to a (non-empty) template, the only expectation was that it either implement an interface or come from a class, so for example, there was no constraint on the class name. To do this, we used the reflectivity of the C# language to find the class that satisfies the conditions in the dynamically loaded application. Because the model guarantees the same structure of the source code to be examined, it is sufficient to query and instantiate the class implementing the given interface or ancestor. The instance created in this way already provides all the functionality required for the task that is relevant to the test.

G. Metric to Determine Student Solutions

In addition to static and dynamic control, various metrics are also measured from the students' source code, so that the instructor can make further suggestions to the student to reduce the complexity of the code and increase its readability. [25] The metrics measured in the source code are based on metrics defined by Visual Studio [26], supplemented by measuring the number of different entries: Comments, Usings, Namespaces, Types, Classes, Methods, Fields, Variables

IV. Evaluation

The model was implemented in C#. The data were given by the homework of OE-NIK SzTF2 course, which were collected from the e-learning system of Óbuda University. The course was attended by a total of 364 students who had to complete 4 homework assignments during the semester, an overview of this is given in Table I. The 1247 source codes uploaded during the semester could be verified by automatic evaluation in just 3 hours, to which is added the time for syntactic rules and unit tests. Overall, a solution can be evaluated in approximately 9 seconds. In contrast, instructors took an average of 30 minutes to complete a task, in addition to which several (not perfectly working) solutions were marked as correct.

The most common compile time errors come from that the student did not complete their homework and uploaded an empty solution file, while the last place is due to errors due to inattention, such as re-declaring a variable. The most common warnings are those that call the programmer's attention to either a used or uninitialized variable, while in minor cases a typo occurs (for example, giving a value instead of an equality check in an expression that will result in a runtime error). Although these are only warnings, they are important feedback to students to further improve their programming skills by eliminating them.

For duplicates, HashSet, as a simpler task, contains a larger number of identical solutions due not to intentional copying but to the feasibility of an easy solution, while in other cases where more complex functionality had to be created, the number remained low and well represented students who had submitted other students' solutions.

In the case of syntactic tests, the problem was that the students did not read the task statement carefully and used (or did not use) the structures included in the task description.

Based on the results of the unit tests, it can be said that few students managed to meet all the criteria, the main reason being that the implementation of a generic type for each homework was expected. However, in the case of generic types, students did not consider that not only a value but also a reference type could be the generic type. That is, some of the tests ran successfully, while others did not. There have also been cases where students have not or incorrectly handled certain code paths or states during the implementation of a function, such as the deletion of two children in the case of BST (binary syntax tree).

TABLE I
STUDENTS' SOLUTIONS AND A SUMMARY OF THE RESULTS OF THE EVALUATION, GROUPED UP BY HOMEWORK TYPE.

Homework	Number of			Criteria			Unit Tests					Nbr of Duplications	
	Solutions	CTW	CTW	C1 pass	C2 pass	C3 pass	Pass all	Failed 1	Failed 2	Failed 3+	Failed all	L0	L2
LinkedList	315 (87%)	30 (8%)	22 (6%)	279	233	287	106 (36%)	30	49	94	14 (5%)	2	2
BST	317 (87%)	6 (2%)	1 (1%)	274	175	306	199 (63%)	26	34	57	0 (0%)	4	17
Graph	309 (85%)	40 (11%)	26 (7%)	245	230	259	38 (13%)	95	12	116	22 (8%)	6	8
HashSet	306 (85%)	39 (11%)	15 (4%)	265	270	233	182 (63%)	0	42	49	18 (6%)	104	124

V. CONCLUSION

Overall, automatic assessment was found to be useful and forward-looking by both the students and the faculty. The biggest benefit for the instructors was that the review process was speed up ($\sim$ 99% time savings) and, thanks to the detailed report, students received more feedback on their submitted work, thus avoiding more general questions. In addition to quick and detailed feedback, independent and unbiased evaluation was the most important thing for the students, thanks to which they could master and complete the subject material more successfully.

For more complex accounting, the model has not been used as an automated evaluation system, because in such cases it is no longer necessary to run only an executable file, but even an underlying database or user interface (GUI), which, although not impossible, is more difficult to verify. It would make it more time consuming to build the evaluation logic than manual checking.

Our further development plans include expanding the rules of syntactic tests, automatically generating a large number and full unit tests for the tasks, which can reveal the faulty code in more detail, and adapting the evaluability of other programming languages taught at the University (Java, PHP, Assembly) to the model.

VI. ACKNOWLEDGEMENT

Supported by the ÚNKP-19-3 New National Excellence Program of the Ministery for Innovation and Technology.

REFERENCES

[1] K. Gábor and H. Éva, "Irisz project: A web application for the introduction of university students to the labor market," *8th International Symposium on Applied Informatics and Related Areas*, pp. 125–129, 11 2013.

[2] F. Prados, J. Soler, I. Boada, and J. Poch, "An automatic correction tool that can learn," *Proceedings - Frontiers in Education Conference*, 10 2011.

[3] D. Barnes, "Objects first with java: a practical introduction using bluej," 01 2006.

[4] R. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*, 01 1981.

[5] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-d tool for introductory programming concepts," *Journal of Computing Sciences in Colleges - JCSC*, vol. 15, 01 2000.

[6] G. Csapó, "Placing event-action-based visual programming in the process of computer science education," *Acta Polytechnica Hungarica*, vol. 16, no. 2, pp. 35–57, 2019.

[7] G. Rößling and B. Freisleben, "Animal: A system for supporting multiple roles in algorithm animation," *Journal of Visual Languages Computing*, vol. 13, pp. 341–354, 06 2002.

[8] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with jeliot 3," *Proceedings of the Workshop on Advanced Visual Interfaces AVI*, pp. 373–376, 01 2004.

[9] A. Korhonen, L. Malmi, and P. Silvasti, "Trakla2: a framework for automatically assessed visual algorithm simulation exercises," 01 2003, pp. 48–56.

[10] S. Edwards and M. Pérez-Quiñones, "Web-cat: automatically grading programming assignments," *ACM SIGCSE Bulletin*, vol. 40, p. 328, 08 2008.

[11] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *ACM Journal of Educational Resources in Computing*, vol. 5, 09 2005.

[12] F. Prados, I. Boada, J. Soler, and J. Poch, "Automatic generation and correction of technical exercises," 09 2020.

[13] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Codewrite: Supporting student-driven practice of java," 01 2011.

[14] K. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer Science Education*, vol. 15, pp. 83–102, 06 2005.

[15] K. Rahman and M. J. Nordin, "A review on the static analysis approach in the automated programming assessment systems," 07 2007.

[16] Y. Liang, Q. Liu, J. Xu, and D. Wang, "The recent development of automated programming assessment," *Proceedings - 2009 International Conference on Computational Intelligence and Software Engineering, CiSE 2009*, pp. 1 – 5, 01 2010.

[17] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling'10*, 01 2010.

[18] D. Souza, K. Felizardo, and E. Barbosa, "A systematic literature review of assessment tools for programming assignments," pp. 147–156, 04 2016.

[19] S. Tung, T.-T. Lin, and Y.-H. Lin, "An exercise management system for teaching programming," *Journal of Software*, vol. 8, 07 2013.

[20] H. Kitaya and U. Inoue, "An online automated scoring system for java programming assignments," *International Journal of Information and Education Technology*, vol. 6, pp. 275–279, 01 2016.

[21] C. Beierle, M. Kulaa, and M. Widera, "Automatic analysis of programming assignments." 01 2003, pp. 144–153.

[22] K. Naudé, J. Greyling, and D. Vogts, "Marking student programs using graph similarity," *Computers Education*, vol. 54, pp. 545–561, 02 2010.

[23] The .net compiler platform sdk. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/. [Accessed: 01-Jun-2020].

[24] Nunit. [Online]. Available: https://nunit.org/. [Accessed: 01-Jun-2020].

[25] H. Ayman and A. Odeh, "Smscqa: System for measuring source code quality assurance," *International Journal of Computer Applications*, vol. 60, pp. 975–8887, 12 2012.

[26] Code metrics values. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019. [Accessed: 01-Jun-2020].