Programmable Questions in Edgar

I. Mekterović *, Lj. Brkić* and V. Krstić**
* University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
** The Fifth Gymnasium, Zagreb, Croatia
igor.mekterovic@fer.hr

Abstract - **Automated programming assessment systems (APAS) are a valuable tool that is growing in popularity, particularly in the field of computer science education. They can provide quick and objective assessment and feedback to the programming assignments – those that receive source code as a response. Most APASs treat code as a black box and employ dynamic analysis to assess code. Dynamic analysis is straightforward, easily implemented, explainable, and works well in most situations. Edgar is a comprehensive, state of the art APAS, that has been used daily and has evolved for the past six years. This paper examines the pipeline used by Edgar to assess programming questions and presents our enhancements to the traditional dynamic analysis - programmable templates and scripts. Templates enable customized question texts based on the programmable model, so that each student can receive personalized variation of the question. Personalized questions are a great way to fight potential academic dishonesty. Scripts are plugged into the assessment pipeline after the dynamic analysis and can override the default grade by examining some other aspect of the program. We also offer our thoughts on upcoming plans to include generic static analysis as we move closer to a unified assessment pipeline.**

Keywords* - *APAS; automated assessment; dynamic analysis; programmable questions, CS education

I. INTRODUCTION

Automated programming assessment systems (APAS) is an information system used in educational environments to (semi)automatically assess students' answers to programming questions. They typically also support other types of questions, such as multiple-choice questions, and provide monitoring and logging of exams, various statistics, and visualizations, etc. Nowadays, they are typically implemented as web applications. A detailed overview of comprehensive APAS features can be found in our previous work [1]. APASs provide fast and objective assessment and feedback but are less capable of producing partial assessments, especially for code that cannot be executed (e.g., does not compile). Manual assessment of code by teachers is still considered the gold standard, but manual assessment is difficult and very time consuming for the teachers. Nevertheless, APASs are being increasingly developed and used (great recent overviews can be found in [2] and [3]), and it is now hard to imagine a larger computer science course that does not use APASs to one degree or another. At the Faculty of Electrical Engineering and Computing, we have been developing and actively using a state-of-the-art APAS called Edgar for six years, and it has become an indispensable part of many computer science courses, relying on it for part or even all of the assessment. For example, in the previous academic year 2021/22, more than 57,000 exams were administered using Edgar, containing approximately 340,000 questions from 19 different courses. In this paper, we present two improvements made to the classic question assessment pipeline. First, we introduced *templates* to enable programmatic question content generation so that each student could obtain a personalized question variation. Second, we enable custom scripts that are appended at the end of the assessment pipeline and can alter the assessment outcome. Although our focus is on programming questions, both concepts can be used on any type of question. In the following chapters, we briefly outline assessment types in APASs and in Edgar, and then present templates and scripts in dedicated chapters, followed by brief discussion and conclusion.

This work was supported by the European Regional Development Fund under Grant KK.01.1.1.01.0009 (DATACROSS).

II. CODE ASSESSMENT IN AUTOMATED PROGRAMMING ASSESSMENT SYSTEMS

Techniques for evaluating program code can be roughly divided into **dynamic** and **static** analyses. Dynamic analysis is much more prevalent, and most APAS bases their assessment on dynamic analysis.

A. *Dynamic analysis*

In dynamic analysis, the program is treated as a *black box* and tested by running it with different input data and analyzing the outputs. For one program, it is possible to define N test-cases (which can affect the sore with different weights), so it is also possible to achieve partial evaluation. However, the disadvantage of dynamic analysis is that the program must be able to run, so programs that cannot be compiled, for example, will not be able to run and will be rated with the worst rating. On the other hand, this approach is widely applicable because the same principle is applied for any programming language, it is only necessary to establish the process of compilation and execution for a language and to ensure that the execution of the program does not have negative side-effects, i.e., it is necessary to execute the program in a *sandbox* - protected and limited environment. Edgar uses a separate code execution system named Judge0 [4] for secure and scalable code execution.

Dynamic analysis mainly tests the functional aspects of the program, but it is also possible to analyze some non-functional attributes, such as efficiency (CPU, time, memory, etc…). However, latter is much more difficult to implement because it is necessary to ensure that all programs have exactly the same resources at their disposal (CPU, disk, ...), which is almost impossible in virtualized

environments, so it is advisable to repeat the analysis several times and statistically process the results.

B. Static analysis

In static analysis, the program is analyzed without executing the program. It is used to check style, syntax errors, various metrics (number of lines of code, cyclomatic complexity, etc.), program design and structure, special properties (e.g., use of an expression), and even plagiarism detection [5]. It should be noted that the manual review of program code written on paper is actually a static analysis performed by a teacher through visual inspection. In regard to assessment and grading, the typical workflow is as follows: source code is parsed to construct an abstract syntax tree, which is then transformed into a graph representation and compared to a set of reference graphs using graph similarity measures. The best fit is used to find the potential differences, and an overall assessment is given. Static analysis is very dependent on the programming language, and it is difficult to implement it uniformly. Typically, the literature contains works focused on a specific programming language. In addition, it is more difficult to design tasks that are checked in this way (e.g., to provide the set of all correct solutions). The field of application of static analysis is smaller than that of dynamic analysis.

Finally, it should be noted that static and dynamic analysis can be combined into a joint - hybrid evaluation method.

III. ASSESSMENT IN EDGAR

Dynamic analysis has been used in Edgar since the beginning. It was initially developed only for the SQL programming language, where the resulting data sets from two SQL commands were compared, and then dynamic analysis was implemented for standalone languages. In both approaches, the program code is separated into three parts: prefix, main part, and suffix. Only the main part represents the solution that is expected from the student, and the optional prefix and suffix allow the teachers greater options in formulating and testing the questions. Namely, before executing the code, those three parts are merged into one, which means that the teacher can inject their own code before and/or after the student's code. Therefore, for example, it is possible to ask a student to write only a function, and the teacher injects his own code that calls and tests that function. Similarly, it is possible to ask a student to submit just one Java class, which can then be instantiated and/or explored using reflection, and thus even non-functional code properties can be evaluated, e.g., whether the class conforms to naming conventions, etc.

Dynamic evaluation of a question is carried out with an arbitrary number of test-cases. Additionally, test-cases can have an arbitrary weight, that is, they can carry a different number of points. The weight is expressed as a percentage that is subtracted from the initial 100% and is called the penalty percentage (PP). Therefore, the correctness is calculated as follows:

$$correctness := \max\left(0\%, 100\% - \sum_{fail} PP(fail)\right)$$

The calculation of the grade itself is carried out in two stages: first, the correctness is evaluated on a scale of 0% - 100% as described, and then the number of points is assigned from the correctness based on the assigned grading model. The grading model is defined as a triplet:

$$gm := (c, i, e)$$

where c, i, and e are values for correct, incorrect and empty answers respectively. As a rule, the incorrect score is set to a negative value for multiple-choice questions to discourage students from guessing. For programming questions, it is always set to zero. Given the grading model, the final score is calculated according to:

$$score = \begin{cases} \boldsymbol{gm.e} & answer\ is\ empty \\ \boldsymbol{correctness} * (\boldsymbol{gm.c} - \boldsymbol{gm.i}) \\ \quad + \boldsymbol{gm.i} & answer\ is\ non\ empty \end{cases}$$

Further details and examples can be found in our previous work [1].

Figure 1. shows the simplified Edgar's dynamic-evaluation pipeline, as used in most APAS-es. Technical details are omitted, and, for the sake of simplicity, it addresses a single question. In reality, an exam consists of an arbitrary number of questions. This model works well, but after several years of use, ideas emerged for some more demanding evaluation scenarios. With this type of organization, all students get the same common question with the same content. If one wants to reduce the possibility of plagiarism, it is necessary to make several similar versions of the same question and then assign them to students by random selection. Although this is possible, such methodology is tiresome for the teacher, error-prone and requires significant additional effort. This problem is addressed by the introduction of **templates** - the possibility to programmatically generate or adjust the text of the assignment for each student. The second improvement that is presented here refers to the possibility to programmatically manipulate the outcome of the assessment performed by Edgar, that is, to override the default assessment at the end of the evaluation pipeline.

Both concepts are described in the following chapters.

IV. QUESTION TEMPLATES

Templates enable the generation of customized question content based on the programming model. They are inspired by the MVC pattern where content is generated in a view based on a model, while the whole process is orchestrated by the controller. MVC frameworks typically generate HTML using programming constructs interleaved with HTML code. The method of interleaving, that is, the syntax that can be used in the view definition depends on the framework and the so-called *view/templating engine* that is used to produce the final HTML. For example, in the .Net MVC framework Razor syntax is used [6], the Node.JS express development framework allows various template engines [7](EJS, handlebars, etc.), etc..

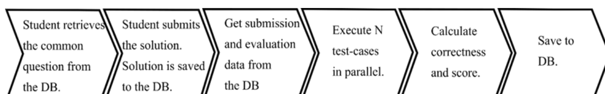

Figure 1. Edgar's initial assessment pipeline

To introduce such a concept into the existing model in Edgar, it is necessary to:

- generate each student's **data object** only **once** when the exam is **first** started (question retrieved)
    - the data object is defined in design time by the teacher using the JavaScript programming language.
- generate the **custom question content**, and
- **store both** the data model and question content in the database

In other words, when the student requests the question, the database is checked for their existing custom question, and if there is no question, the above-mentioned steps are taken. Ultimately, in both cases the question is retrieved from the database. The concept is illustrated with a simple example: Figure 2. and Figure 3. show the definition of a templated multiple-choice question where the assignment is to add two numbers. The model (data object) here consists of two random numbers x and y and three answers: a1, a2 and a3, with a1 being the correct answer:

Template data object

```
{
  init() {
    this.x = this.randomInt(13, 37);
    this.y = this.randomInt(133, 337);
    this.a1 = this.x + this.y;
    this.a2 = this.a1 - 1;
    this.a3 = this.a1 + this.randomInt(1, 5);
  }
}
```

```
{
  "x": 27,
  "y": 188,
  "a1": 215,
  "a2": 214,
  "a3": 216
}
```

Figure 2. Template data object definition and a random instance

The data object is constructed by invoking the `init()` method, which produces some random values, such as the one shown in Figure 2. These variables can then be used both in question text and answers, as shown in Figure 3. :

Question (GitHub flavored markdown)

```
### Calculate {{x}} + {{y}}?
```

Answers: a {{a1}} 100; b {{a2}} 100; c {{a3}} 100; d I don't know 100

Preview: Render template

Calculate 27 + 294?

Answers: a 321; b 320; c 324; d I don't know

Figure 3. Template definition (left) and preview (right). Tempalate references variables x and y from the data object.

Note the "moustache/handlebar" syntax in the question definition (upper left part of Figure 3.) - string interpolation is performed with double curly braces. Edgar uses GitHub markdown for rich text content and Handlebars.js as a templating engine. Handlebars (and consequently Edgar) support more than just inserting values; it supports expressions, loops, etc. [8]. The template is rendered in the context of the given data object. The right part of Figure 3. shows the rendered template using the ad hoc instantiated data object every time the "Render template" button is clicked. The data object is constructed using a three-level hierarchy:

- **global** data object (same for all courses, cannot be changed by a teacher)
- **course** data object (can be changed by the teachers in the course)
- **question** data object (defined when the question is defined)

Global and course data objects are meant to store utility methods and variables to simplify the question's data object source code (such as the `randomInt()` method which is not part of JavaScript). For instance, if global, course and question data objects are:

Global:

```
{
  randomInt: function(minInt, maxInt) {
    return Math.floor(Math.random() * (maxInt - minInt + 1)) + minInt;
  }
}
```

Course:

```
{
  randomBoolean: function() {
    return !!this.randomInt(0, 1);
  },
  currAcYear: "2022/2023",
  someOtherConstant: 42
}
```

Question:

```
{
  init() {
    this.x = this.randomInt(13, 37);
    this.y = this.randomInt(133, 337);
    this.a1 = this.x + this.y;
    this.a2 = this.a1 - 1;
    this.a3 = this.a1 + this.randomInt(1, 5);
  }
}
```

Then, the final data object is assembled as follows:

```
{
  randomInt: function(minInt, maxInt) {
    return Math.floor(Math.random() * (maxInt - minInt + 1)) + minInt;
  }
  randomBoolean: function() {
    return !!this.randomInt(0, 1);
  },
  currAcYear: "2022/2023",
  someOtherConstant: 42
  init() {
    this.x = this.randomInt(13, 37);
    this.y = this.randomInt(133, 337);
    this.a1 = this.x + this.y;
    this.a2 = this.a1 - 1;
    this.a3 = this.a1 + this.randomInt(1, 5);
  }
}
```

Which when init() is called, will produce e.g.:

```
{
  "currAcYear": "2022/2023",
  "someOtherConstant": 42
  "x": 31,
  "y": 203,
  "a1": 234,
  "a2": 233,
```

```
    "a3": 239
}
```

V. OVERRIDE SCRIPTS

Scripted questions enable teachers to append custom scoring codes/scripts at the end of the assessment pipeline. Scripts were inspired by *checker* programs in competitive programming: "In programming competition environment, a checker is a program written for the purpose to check the output of the contestant's program for a task that has many solutions. Usually, a checker is written manually as needed."[9]. The purpose of checkers, or *override scripts* as they are called in Edgar, is to enable custom grading in advanced scenarios. Consider the following example assignment:

> **Assignment:**
> Print "hello world" in C programming language without using the `printf` function.

The output produced by the student's solution – "hello world" can be checked with dynamic testing, but the *non-functional* program property "without using the printf" cannot. However, a script that would have access to submitted code could simply search for "printf" in the code and override the score if "printf" is found. If not found, the score should be inherited from the dynamic analysis. Obviously, the script must have access at least to code and dynamic analysis score.

Accordingly, the scripts are embedded at the end of the pipeline, as shown in Figure 4.

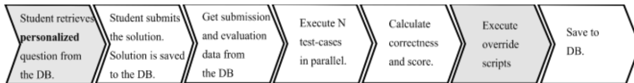

Figure 4. Assessment pipeline in Edgar with newly added features (gray): templates (at the beginning) and override scripts (at the end).

If scripted questions are used, the assessment takes place as follows:

- Edgar performs the built-in default assessment and scoring and forwards all the relevant data further down the pipeline.
- A custom evaluation script written by the teacher is executed:
  - The script has all the data from previous steps at its disposal (question definition, student's answer, evaluation score from dynamic analysis, etc.)
  - The script acquires the data object context. If templates were not used, then the data object is constructed from the course and question data object, thus having utility methods at disposal.

The script must implement the `getScore()` method which returns **the score object**. Thus, the teacher can alter the default score object that was received as an argument.

Figure 5. shows the script for this example and Figure 6. shows the resulting score object. Note that scripts can be used not only to modify the score object, but also to provide helpful hints to students. Actually, having all the data at their disposal, the teacher can program anything.

The script can be written in arbitrary programming language (supported by Edgar, e.g. C, Java, Python…) or in JavaScript, with the difference being:

- JavaScript (recommended) script is evaluated and executed *in-process*, at the web-server machine serving the request. All relevant data is embedded in the script data object and is available simply as `this.something`, no parsing or evaluating necessary.
- Other programming languages (Java, C, …) - scripts are being sent to the Judge0 code execution engine – the very same used to evaluate students' code. This means that Edgar invokes an HTTP request to Judge0, where the code is compiled and executed in the sandbox. Relevant data are serialized as JSON and sent to the script as the single stdin argument. The script/program must deserialize the input to do anything useful. The program must return a new score, again - serialized as JSON, by writing that string to stdout. Edgar subsequently deserializes the stdout and proceeds.

Custom scoring script

```
1  {
2      getScore() {
3          let answer = this.getCodeAnswer(); // helper method from the global data object
4          answer = this.removeComments(answer); // also, helper method from the global data object
5          if (answer.indexOf("printf") >= 0) {
6              return this.getIncorrectScore("You cannot use printf"); // helper method to construct the score
7          } else {                                                    // according to the grading model
8              this.score.hint += " I was also here at " + (new Date()).toISOString();  // just for fun
9              return this.score;
10         }
11     }
12 }
```

Figure 5. Override script checking for usage of printf function. If found, an incorrect score is returned. Otherwise, the score is unchanged, and the timestamp is appended to the hint.

```
Score is:
{
  "is_correct": true,
  "is_incorrect": false,
  "is_unanswered": false,
  "is_partial": false,
  "score": 1,
  "score_perc": 1,
  "hint": "Correct. Well done! I was also here at 2023-01-31T09:29:02.893Z",
  "c_outcome": [
    {
      "input": "1",
      "percentage": "100.00",
      "hint": "Correct. Well done!",
      "output": "hello\n",
      "expected": "hello",
      "mode": "check elements order : false, case sensitive : false, ignore whitespace : true",
      "isCorrect": true,
      "comment": null,
      "stderr": null,
      "is_public": false
    }
  ]
}
```

▸ Click to see the entire object...

Figure 6. Score object produced by the script in Figure 5. for correct submission.

For all these reasons, remote procedure calls being dominant, JS scripts are much faster and easier to write. On the other hand, in certain advanced scenarios, where override scripts need complex libraries to check the program (e.g., NumPy, OpenCV, etc.) the communication overhead is well worth the cost.

Both templates and override scripts can be used on any type of question, not just programming questions. Our final example shows a combination of templates and scripts on a **free-text** question. The assignment is a classic first-year programming task – provide a hexadecimal representation of a number according to the IEEE 754 standard. First, to provide each student with their custom number, we define the data object as follows:

```
{
    toHex(double) {   // convert number to IEEE754 hex
      const BYTES = 4;
      const buffer = new ArrayBuffer(BYTES);
      const float32Arr = new Float32Array(buffer);
      const uint32Array = new Uint32Array(buffer);
      float32Arr[0] = double;
      const integerValue = uint32Array[0];
      const integerBitsHex = integerValue.toString(16);
      return integerBitsHex;
    },
    init() {
        this.x = this.randomInt(100, 200);
        this.x += this.randomBoolean() ? 0.5: 0;
        this.x += this.randomBoolean() ? 0.25: 0;
        this.x += this.randomBoolean() ? 0.125: 0;
        this.x += this.randomBoolean() ? 0.0625: 0;
        this.hex = this.toHex(this.x);
        this.correct = this.hex.toUpperCase();
    }
}
```

```
{
  "x": 143.8125,
  "hex": "430fd000",
  "correct": "430FD000"
}
```

Figure 7. Question data object definition and one random instance. The object provides the random decimal number and corresponding hexadecimal IEEE754 representation which will be required of the student.

The `init()` method chooses a random integer in the [100, 200] range and then randomly adds a few negative powers of two so that the decimal number converts nicely to binary (unlike. e.g., 0.3 which has an infinite number of decimals). The program also immediately calculates the correct answer and stores it in the data object as the "`correct`" variable. The template is defined simply as:

> Provide a hexadecimal representation of the number {{x}} according to the IEEE 754 standard.

which will render different numbers for different students. Finally, the override script must check the student's answer and compare it with the prepared correct answer in the data object (Figure 8.). Figure 9. shows a simplified UML sequence diagram of the assessment pipeline for questions using both templates and override scripts.

```
{
    getScore() {
        this.answer = this.getFreeTextAnswer().toUpperCase();
        this.answer = this.removeWhitespace(this.answer);
        if (this.answer === this.dataObject.correct) {
          return this.score;  // free texts are always correct by default
        } else {
          return this.getIncorrectScore("Failed: " + this.answer
              + "<>" + this.dataObject.correct);
        }
    }
}
```

Figure 8. Student's answer is trimmed and converted to uppercase and compared to the correct answer.

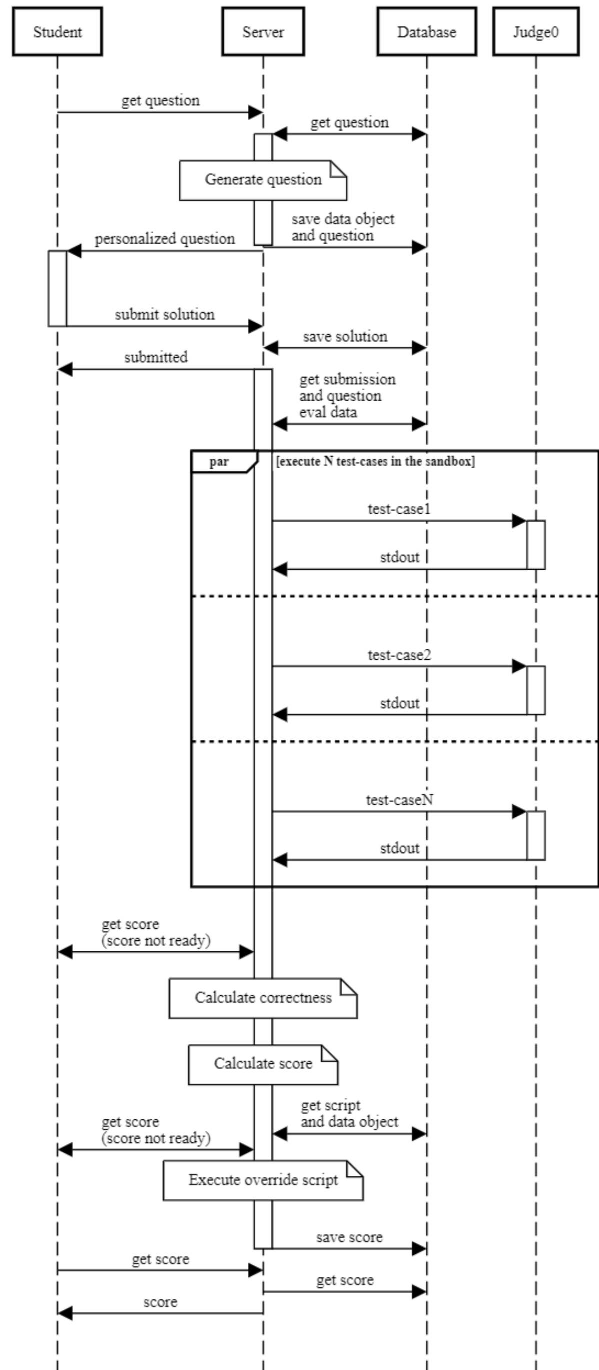

Figure 9. Simplified UML sequence diagram of the assessment pipeline for questions using both templates and override scripts

Override scripts enable teachers to parse free text answers and grade them. In this way, the teacher can prescribe the allowed nomenclature and then programmatically interpret the submitted solutions and evaluate them. Since textual representations can be prescribed for many data structures (graphs, tuples, tables, lists, etc.), this approach opens up great possibilities for (semi)automatic testing. If necessary, the grades assigned in this way can be manually reviewed and changed later, which is also supported in Edgar. We successfully use this type of assessment, where automatic assessment is followed by teacher control, in several courses at FER. Of course, automatic evaluation is of great help to the teacher, especially for correct solutions. In fact, most of the time, it is necessary to review only the answers that were declared incorrect by the automatic evaluation.

It should be noted that override scripts could be used to perform additional static analysis on the submitted code. In this way, it is even possible to create a hybrid model - static and dynamic analysis together. However, we believe that static analysis via override scripts would not be a good user experience for a teacher who would have to do too much programming. In terms of future development, our plan is to:

- Develop a separate standalone component that can perform static code analysis and return results in a standard format (e.g., SARIF [10])
- Allow question authors to include static analysis in the evaluation pipeline, and then evaluate the solution using a hybrid model or static analysis only.

In other words, our goal is to provide a unified assessment pipeline where teachers can opt-in for various assessment features and combine them in configurable ways to provide the assessment. As a side note, although we use term “pipeline”, technically some steps are performed in parallel for performance reasons. Of course, these facilities can be used in other settings, not only during grading. For example, in e-learning environments, the student would be helped and guided how to fix the code and come up with a correct solution.

VI. Conclusion

In this article, we introduced two enhancements to the standard procedure for creating and evaluating questions in automated code evaluation systems: programmable templates and override scripts. Templates allow teachers to programmatically generate question content. In this way, a single question definition can yield multiple variations and students can receive personalized questions. Scripts allow the teacher to build upon the default grade and programmatically assign a final grade, or just a helpful comment. In addition to programming questions, both mechanisms can be used for other types of questions, which gives teachers much more freedom and opportunities in composing questions.

References

[1] I. Mekterovic, L. Brkic, B. Milasinovic, and M. Baranovic, “Building a Comprehensive Automated Programming Assessment System,” IEEE access, vol. 8, pp. 81154–81172, 2020, doi: 10.1109/ACCESS.2020.2990980.

[2] J. C. Paiva, J. P. Leal, and Á. Figueira, “Automated Assessment in Computer Science Education: A State-of-the-Art Review,” ACM Trans. Comput. Educ., vol. 22, no. 3, pp. 1–40, Jun. 2022, doi: 10.1145/3513140.

[3] S. Combéfis, “Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools,” Software, vol. 1, no. 1, pp. 3–30, 2022, doi: 10.3390/software1010002.

[4] H. Z. Dosilovic and I. Mekterovic, “Robust and scalable online code execution system,” 2020 43rd Int. Conv. Information, Commun. Electron. Technol. MIPRO 2020 - Proc., pp. 1627–1632, Sep. 2020, doi: 10.23919/MIPRO48935.2020.9245310.

[5] K. M. Ala-Mutka, “A Survey of Automated Assessment Approaches for Programming Assignments,” http://dx.doi.org/10.1080/08993400500150747, vol. 15, no. 2, pp. 83–102, 2007, doi: 10.1080/08993400500150747.

[6] “Razor syntax reference for ASP.NET Core | Microsoft Learn.” https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-7.0 (accessed Jan. 31, 2023).

[7] “Template Engines.” https://expressjs.com/en/resources/template-engines.html (accessed Jan. 31, 2023).

[8] “Handlebars.” https://handlebarsjs.com/ (accessed Jan. 31, 2023).

[9] R. I. Hadiwijaya and M. M. Inggriani Liem, “A domain-specific language for automatic generation of checkers,” Proc. 2015 Int. Conf. Data Softw. Eng. ICODSE 2015, pp. 7–12, Mar. 2016, doi: 10.1109/ICODSE.2015.7436963.

[10] “SARIF Home.” https://sarifweb.azurewebsites.net/ (accessed Jan. 19, 2023).