Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

Effects of automated test case generation on the evaluation of student assignments

Supervisor:
Máté Cserép
Assistant Lecturer

Author:
Marcell Ferenc Hajdu
Computer Science MSc

Budapest, 2023

Topic declaration

The goal of my thesis is to ascertain the effects of the introduction of automatically generated test cases in large numbers alongside the already existing manually written test cases which are usually small in number. To achieve this, I will expand the TMS assignment evaluator and grader system with a test case generator. With this I will be able to generate test cases in large quantities, furthermore this will provide an option for teachers to generate test cases automatically in the future. Using this new functionality on the tasks of previous semesters I will generate new test cases which can be used to re-evaluate the student submissions for the respective tasks. Finally, I will compare the already existing evaluations with my newly created ones. In conclusion the subject of this research is to find out if there are any differences between the new and old evaluations, and if so, is it possible to filter out the not completely correct student submissions.

Contents

Chapter 1

Introduction

Evaluating student assignments in higher education is always tedious, primarily because of the ongoing boom of Computer Science majors that nearly tripled from 2009 to 2015 and doubled from 2011 to 2015[1]. This increase in students causes that assignment evaluation is taking up a sizeable portion of an educator's work time. However, as technology advances, automatic assignment evaluation has become more and more accessible and accurate, so nowadays, it is usable in an academic setting. Now we arrived at another stepping stone into fully automated assignment evaluation, and that is generating test cases for student assignments automatically. The goal of this method is to create a comprehensive test suit that can cover a good percentage of the assignment under test to measure its completeness. Most of these techniques require the end user to create a specification or guidelines to specify the behavior of the test case generation; in some cases, an example solution or a so-called oracle might also be necessary. Leveraging the usefulness of this additional level of automatization is worth evaluating in an educational setting, as it will significantly improve the uniformity of test cases for student assignments and potentially reduce the burden on educators. So, in general, I hypothesize that this technology can improve the evaluation process of student assignments, primary console applications.

Despite the apparent potential of this technique, there is still a need to investigate automatic test case generation, particularly in an academic context. Furthermore, because of the wide variety of potential test case generation techniques originating from the numerous testing methods, examining such systems' overall effectiveness and approachability is necessary, as one technique can give widely different results than another based on the runtime context and implementation. Therefore, addi-

tional research is imperative to accurately assess the potential of automatic test case generation in higher education, particularly in Computer Science courses.

This study aims to answer some of these questions and provide a framework for future studies. The main research questions are as follows:

1. Does automated test case generation change the evaluation results of assignments?

2. Can automated test case generation find edge cases like manual tests?

3. Does automated test case generation improve the evaluation process for educators?

Alongside attempting to answer these questions with a thorough investigation and provide empirical proof for them, I aim to provide a general framework from which, with minimal modification of the testing techniques used during the test case generation, further comparative studies can be based to ascertain the most optimal underline methods to optimize test case generation in an academic setting.

The Thesis is divided into five chapters to delve into this topic in greater detail. First, I am starting an in-depth literature review in Chapter 2, where I compare some already existing student assignment evaluation methodologies and show the variety of testing techniques, with a great emphasis on the black-box methodology. After that, in Chapter 3, I provide the structure of my research, the data-gathering method, and the framework for implementation. In the following Chapter 4, I am describe Eötvös Loránd University's Task Management System (TMS), into which I integrated my prototype solution, alongside going into more detail about the specifics of my implementation. Second to last, I am going over my findings in Chapter 5, where I detail the empirical findings. Lastly, Chapter 6 provides a comprehensive overview of the study and proposes possibilities for further research.

Chapter 2

Related works

While researching related works of literature, my main goal was to explore two sets of topics.

1. Firstly, the outlook of already existing automatic assignment grader systems and their main techniques and approaches. Given the circumstances of the experiment and the example code created for it, it is imperative to examine assignment grader systems thoroughly.

2. Secondly, the different types of testing, like black and white-box testing, model, or input-based testing, are required to explore options for improvement in test case generation and testing and for the approach for creating an example application.

So the following related work section will dive into these topics and discuss the information gathered from relevant literature.

2.1 Existing Automated Grading Systems

Automated grading systems became increasingly relevant with the emerging online education sphere and the continuously growing traditional education institutions. It speeds up the time student submissions are graded, provides helpful early feedback for students, and frees up educational resources. In this section, I will take s look at some papers that discuss this topic.

According to "Automated Grading Systems for Programming Assignments: A Literature Review" by Aldriye, AlKhalaf, and Alkhalaf [2], automated grading sys-

tems can be categorized into three categories based on the level of their automatization of grading: automatic, semi-automatic, and manual. While completely automatic systems provide a generally comprehensive functionality, because of their complexity, they also need more feedback readability, UI/UX usability, more variety of programming languages, and in some cases, localization. On the other hand, semi-automatic systems provide generally more understandable feedback, as teachers and peer reviewers write it, but also take up more time. Lastly, manual systems focus on the ease of upload and feedback giving but provide no automatization[2, 3, 4].

There are generally three techniques used in automatic programming assessments.

1. The first one is a static approach. One specific approach that comes from this static method is where the student's and instructor's programs are compared non-structurally, usually by translating it to some common intermediate language or pseudo code[5, 6].

   Another static option is analyzing the code with a static doe analyzer to find syntax violations, security vulnerabilities, undefined values, and coding standard infractions. This approach usually uses graph-based techniques where a graph represents the code[5] or Hoare logic can also be used[6].

2. The Second alternative is a dynamic strategy. Here the code is executed to assess errors and the design of the program[5]. Usually, the execution of the program uses some test data that can be provided or generated. With this method, we can already rule out serious failures like compilation errors or sometimes even security threats[6]. Generally, there are two approaches to dynamic testing. First is withe-box testing, where information on the inner working of the program under test is needed, like unit testing. The second is black-box testing, where we asses the functional behavior or the input-output relation of a program[5, 6, 7].

3. The last techniques used in automatic programming assessments are a hybrid approach where static and dynamic analysis is applied to overcome the shortcomings of both strategies[5].

2.2 White-Box Testing Techniques

Withe-box or structural testing techniques are designed based on information derived from the source code. Therefore, we understand how the code should work and write tests based on these assumptions. The goal of the test cases is to create paths or branches that the code must cover so we can ensure their proper execution. In order to further refine the effectiveness of tests, we set up criteria like path coverage and branch coverage. Programmers are the ones who usually use white-box testing at a low level to find logical errors and misconceptions or for debugging[8, 9]. We can generally group white-box testing techniques into static and structural white-box testing.

Static white-box testing ensures the source code complies with functional requirements and coding standards to see if all functionalities and error handling are covered. These techniques only involve the source code, so the program is not compiled or executed. Usually, static white-box testing requires someone with a high level of technical knowledge. In case of a Code walkthrough, a knowledgeable colleague of the code's author reviews the new code to search for possible errors and ask questions about the program. While during a formal inspection, we organize a series of meetings to understand the code's functionality, find problems, and fix them[8][9].

Structural white-box testing builds upon the code, the structure of the program, and its design. We can categorize these into four general groups: Coverage Testing, Basic Path Testing, Loop Testing, and Data Flow Testing[10, 9].

Firstly the most basic form of coverage testing is statement coverage testing, where each statement is covered at least once. A different type of coverage test is branch coverage, where if you think of a statement as a node, the technique aims to traverse each edge between nodes at least once as displayed in Figure 2.1. Similarly to branch coverage, decision coverage aims to test all decision branches, boolean conditions, and boolean assignments at least once. Finally, function coverage aims to test all functions of the code[11, 12].

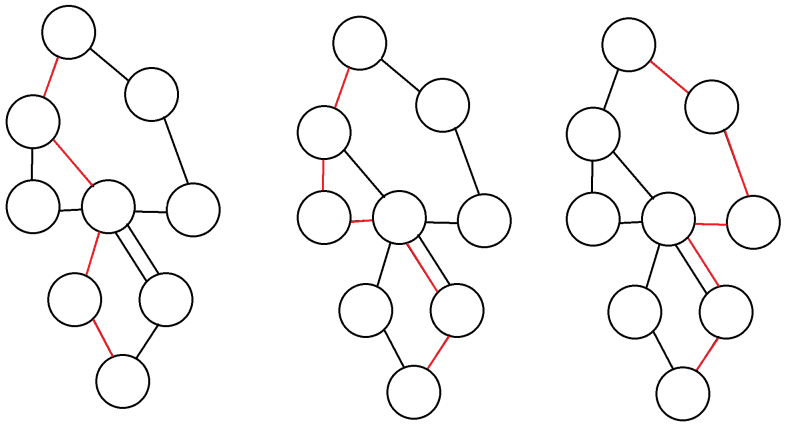

Figure 2.1: Flow Graph for Branch Coverage.

One of the Basic Path Testing techniques is cyclomatic complexity. This complexity measures the logical difficulty of a program based on its Control Flow Graph, a directed graph where a node with a condition is called a predicate node, and edges that start from a predicate node must converge in a predetermined node. The Control Flow Graph has five notations to bild from as shown in Figure 2.2. Loop testing builds on the simple fact that loops are necessary for many algorithms, and the beginning and ends of loops are where many errors occur. The main goal of this testing is to determine the validity of loops and ensure that the program does not enter an infinite loop. Lastly, data flow testing verifies the lifecycle of data in the program.[9][13]

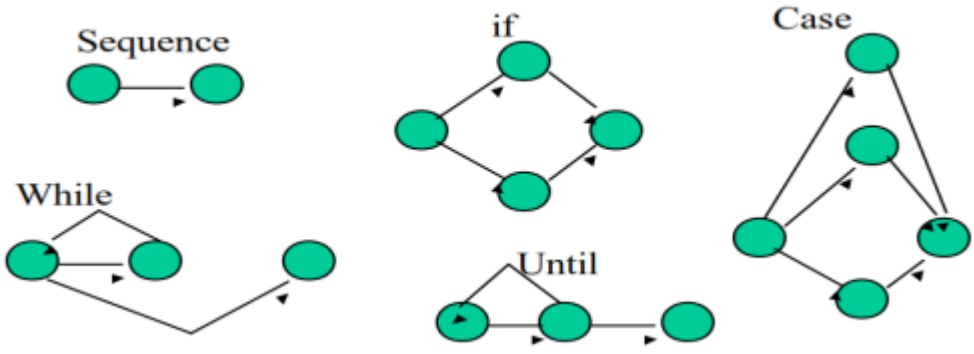

Figure 2.2: Different control Flow Graph Notations. *Note: reprinted from Black Box and White Box Testing Techniques - A Literature Review by Nidhra [9]*

2.3 Black-Box Testing Techniques

Black-box testing involves a range of testing techniques that aim to evaluate the program under test from the perspective of a customer or end-user. This approach identifies and assesses potential issues or errors that may arise during real-world usage, providing valuable insights into the program's overall functionality and performance. Organizations can guarantee that their software meets the highest quality and usability standards by prioritizing the customer's needs in their testing approach. This results in increased customer satisfaction and improved business performance. The tester only knows the input and the expected outcome[9]. We may consider many positives like there is no inherent creator bias during black-box testing as the tester has no information on how the program works and the ease of creating test cases because the testers only concern themselves with the functionality and no inherent programming knowledge is required. Lastly, this testing generally performs better on a larger codebase.

Also, some negatives are up for our consideration. For example, the maintenance of test suites is a complex task, as well as the testers also need precise specifications for adequate testing. Furthermore, they are only guaranteed to test some program branches, as complete coverage is not impossible, but it is unlikely[14].

When it comes to black-box techniques testing, there are four main approaches that we can classify them. These approaches include random testing, model-based testing, testing with complex inputs, and combinatorial interaction testing. These are different approaches to testing, each with its strengths and weaknesses. To decide the choice of approach, we should consider the specific requirements of the testing process[15].

Random testing is one of the earlier researched techniques neglected in studies in the last few decades because of its perceived inadequate performance. However, recently with improvements in empirical research and heuristics, new advancements have been made[15]. This technique provides a high degree of automation and ease of use[16].

Testing with complex inputs is a recent method that has interested researchers. Of course, research on syntactically complex inputs was conducted from many angles throughout the years. However, this new wave of study's central premise is to create syntactically complex inputs that should also be semantically accepted[15].

Model-based testing is a highly effective approach to black-box testing that is only limited by the cost of creating detailed models. However, recent studies on generating these models might prove a way to mitigate the costs mentioned above[17][18][15].

Combinatorial interaction testing focuses on reducing the coverage of all combinations of test inputs. The technique aims to achieve this by only covering a portion of combinations, as most errors occur in a few input combinations[15][19][20].

In the following section, I will write about various black-box testing techniques, not just general categorizations:

Equivalence partitioning is a black-box testing technique that divides the input data into segments that helps the reduction of the number of test cases[14]. A specialized version of equivalence partitioning is equivalence class partitioning, where the program behaves the same for each input in a partition class. The process of creating equivalence classes involves a thorough analysis of the input. This analytical approach identifies various similarities and patterns to group the input to form equivalence classes. In addition, this technique enables a clear and concise data categorization, making it easier to process and understand its underlying structure. With the help of equivalence classes, only one test case from each class is required, so the number of test cases can decrease rapidly. The difficulties lie mainly in identifying the correct equivalence classes[9][21].

Cause-effect graphs are directional graphs where an input condition is the cause, and the performed actions are the effects. These input conditions form a node in the cause-effect graph, and the connections between them describe the causal effect of these input conditions. A cause-effect graph has four basic symbols: identity, negation, logic OR, and logic AND[14] as displayed in Figure 2.3.

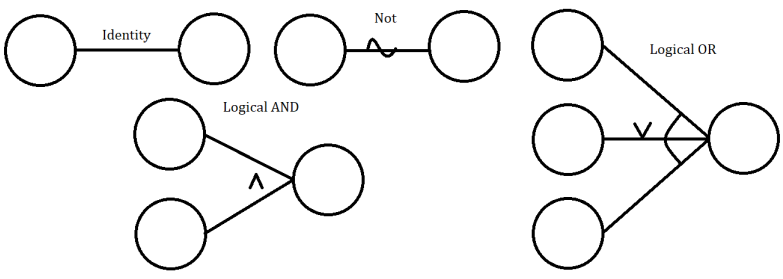

Figure 2.3: Basic elements of cause-effect graphs

Boundary value testing's central tenet is that bugs tend to congregate at the boundaries. This means that errors tend to be found more near the boundaries of equivalence classes, so logically, the tester needs to emphasize values that are maximums, minimums, just inside, and just outside of the boundaries[14][21].

Orthogonal array testing is a technique where each column represents a variable factor, and each row represents a test case. Instead of representing all possible combinations of factors and levels, it combines factors pair-wise[14]. Some benefits of orthogonal array testing are reduced testing time, minor test suits, and uniformly distributed test coverage as shown in Figure 2.4, but this method only guarantees a limited program coverage.

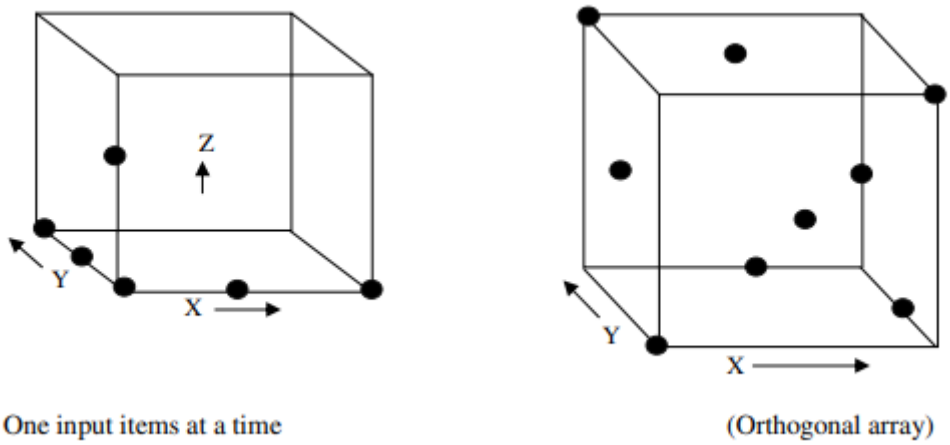

Figure 2.4: Geometric view of test cases. *Note: reprinted from Different Approaches To Black box Testing Technique For Finding Errors by Khan [14]*

Pure random testing is the simplest form of random testing. It works by generating independent random inputs, and the only requirement is to know the interface of the system under test. The main benefit of pure random testing is its low cost compared to other techniques, although its effectiveness was questioned by many. However, more studies have recently discussed the performance of pure random testing, and the results are promising. For example, in one study that used more than 6000 Java classes and previous empirical studies, a correlation of 80% was achieved using the Michaelis–Menten equation to asses the distinctive failures. In another study, a nontrivial lower bound for the test case was found and showed that the best case concerning performance is when all test cases have the same coverage probability. Also, it was

found that there exists a size after random testing became more cost-effective than other approaches[15, 22, 23, 24].

Guided random testing is a technique that combines randomly generated inputs with heuristics and additional information about the system under test to achieve a higher level of effectiveness. A newer enactment of guided random testing is swarm testing. In swarm testing, the primary idea is only to use a portion of the functions to generate test cases. With this, it is more likely to reveal faults that are hard for pure random testing, like memory leaks[15][25].

Adaptive random testing's introductory statement is that diverse test cases will more likely reveal errors than similar ones. The simplest version of adaptive random testing is where a normal distribution is used to select the test cases, so ones that do not have the designated distance from the already generated test cases will be removed. Although this method creates more varied test cases, it also adds additional costs compared to pure random testing. Hence, the latest research in this field focuses on improving the performance of test case generation. A solution for better performance came in the form of test profiles where already executed test cases are the source of the distribution, which is then used to create the following test case[15][26][27].

Model-based testing from inferred models was developed for the apparent purpose that models are generally hard to maintain and sometimes unavailable. Furthermore, inferred models are models that only describe the behavior of the system under test, not its intended behavior, so they cannot be called accurate application models. Nevertheless, they can still be used effectively[15].

Model-based testing from ripped models' aim is to extract a model from the system under test to generate test cases from the extracted model. The extraction of the model can be done in various ways, like analyzing the code or examining the execution traces after running the system under test. Generally, these extracted models define an infinite number of behaviors. Hence, the tester needs to define a test selection criterion, creating a finite set of test cases with a certain level of confidence in correctness. Some notable rippings are GUI ripping, where the extracted models came from the GUI, usually by trying out the GUI events like button clicks. Another interesting ripping is the Event

Semantic Interaction Graph which is similar to Event-flow graphs with the difference of semantics regarding dependencies between events[15][17].

Learning-based model-based testing aims to solve the problem of ripped models, that is, the incompleteness of the model. It achieves this by expanding the model with the generated test cases, which will be used to generate more test cases iteratively[15].

Model-based testing from UML models is a technique that uses UML or Unified Modeling Language, which is a visual representation of a program's structure, behavior, and architecture, to generate test cases. Recently, there have been many studies into using other types of diagrams for a generation, like sequence diagrams and use case diagrams[15][28][29].

Model-based testing from feature models depends on using feature models for test case generation. This model contains features and their dependencies of programs; feature models are usually used to describe a set of software similar to their code yet still considered different because of minor variations[15][30].

Complex Syntactic Input generation involves defining the input structure and using various algorithms to create inputs that meet the specifications. These algorithms and specifications vary from case to case. There are many approaches to this, like recursive methods, which use tree-like structures or dynamic programming, and another method uses simple multidimensional numeric input spaces[15][31].

Complex semantic input generation focuses on realistic and coherent test data creation. Realistic in the sense that the created data should be semantically correct, while coherent in the sense that the generated inputs should be formed of semantically related elements[15].

Chapter 3

Methodology

In this chapter, I present the methodology I used to examine the effects of automated test case generation on student assignment evaluation. Mainly I focused on the potential impact of generated test cases using data from the Faculty of Informatics at Eötvös Loránd University to collect and analyze empirical data to conclude the potential ramifications of this method and provide a possible framework for implementation.

My hypothesis in this study is that the introduction of automated test case generation increases uniformity and saves valuable time on the educator's part. For the purpose of testing these assumptions, I set up the following research questions:

1. Does automated test case generation change the evaluation results of assignments?

2. Can automated test case generation find edge cases like manual tests?

3. Does automated test case generation improve the evaluation process for educators?

To answer these questions, I used a quasi-experimental design and created two data groups. One for a control group consisting of previously submitted student assignments, and the experiment groups made up of the same set of data but tested on the automatically generated test cases. In the subsequent sections, I describe the participants alongside the data collected, the test case generation strategy, data analysis, and the ethical consideration of the study.

3.1 Automated Test Case Generator Design

3.1.1 Requirements

Test data generation for student assignment tester and grader systems can be challenging, primarily because generating various inputs to various assignments causes a dilemma of how much of a teacher's time it takes to operate the generator against the time to create manual tests. The number of unique assignments can also provide difficulties in creating a general specification for faster iteration time. This level of variety is the main reason for providing the requirements for the implemented program and the experiment.

Assignments under test: In an academic setting, the instructor can define various input sources for a student assignment, like a graphical user interface, reading of files, web requests, and inline arguments. For this thesis, I defined a criterion for reducing the scope of possible input sources to assignments that use the standard input for reading their inputs. These assignments are fundamental for educating new computer science students, and I concluded that the scope should be sufficient. Also, these programs must put their outputs into the standard output. So, for the assignment under test, the technique discussed in this thesis expects a console application.

Input restrictions: There can be all kinds of inputs to a console application, such as integers, strings, specific strings, aka enums, guids, and URLs, to name a few. I deemed it necessary to limit the possible number of inputs to these three: integer, string, enum

Additional information on each type is necessary for further clarifications, such as an upper and lower bound for integers, a length for strings, and a list of strings for the enum.

Generated inputs: Input tuples will be generated based on the input restrictions. A value between the upper and lower bounds for an integer will be created. In the case of a string, a randomly generated string with a given length is generated. Lastly, about an enum, a randomly selected enum member is to be generated.

Example teacher solution: To facilitate simple output generation, the teacher responsible for creating the task must provide an example solution. This example solution will be our oracle that will provide a matching output for the generated inputs. All example solutions should also abide by the requirements applied to the assignments under test.

End-user experience: As previous studies showed [2], one of the biggest hindrances to adopting an automatic tester and grader system, and as a consequence, the underlying generator is a hard-to-understand GUI. As such, the primary goal is a simple, intuitive, fast end-user experience where users make changes efficiently and responsively.

Test case generation expectations: Test case generation should create various input-output pairs to cover many possible failure points. Dealing with a complex assignment involving multiple test cases can be daunting, especially considering the numerous potential outcomes. While it is technically possible to cover every combination, the sheer number of possibilities makes it impractical to do so, mainly because the generation process should be manageable for the test runner system.

3.1.2 Design Principles

During the design phase of the test case generator, it was crucial to prioritize modularity and scalability as critical factors. The employed methodology guaranteed that the system was developed with a significant degree of versatility and adjustability, facilitating the smooth integration of additional software elements. Furthermore, implementing a modular and scalable architecture enhanced the system's functionality and performance, thereby leading to improved productivity and efficiency. Therefore, the Generator should consist of three main parts: An Input Generator, an Output Generator, and the Main Generator to orchestrate the generation as displayed in Figure 3.1.

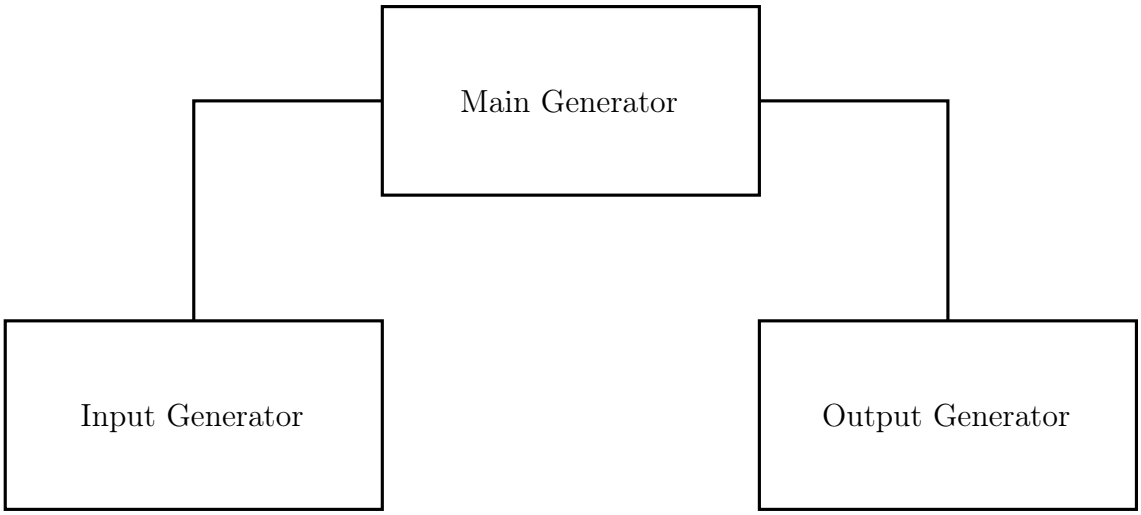

Figure 3.1: Fundamental design of the Test Case Generator

Responsibilities of modules

The Input Generator parses the incoming data as specified in the Requirements section. Then it should start the generation process based on the required data type. During this process, many liberties can be taken to finetune the test case generation for one need.

For example, there are many string generation methods, from scrambling a list of characters to something more complex like a cryptographic key generation. For this study I choose the cryptographic key generation as it produces more reliable and unique results than simple shuffling.

It is a relatively straightforward approach for enums, as it only entails choosing an element from a list.

The most variation should come from the selection of an integer. One can choose many approaches, such as choosing a boundary value or using some heuristics to determine the integer. Nevertheless, I decided on a Pure Random method with performance in mind. Also, It is expected, from the test case generator to create many test cases, so Pure Random testing should yield satisfactory results.

The Input generator is also responsible for pairing the generated inputs. There are various ways to do this, like all-pair matching or orthogonal array testing. However, I decided on a Pure Random approach to go along with the philosophy of building from the most straightforward solutions while approaching a more complex outcome.

The Output Generator is responsible for generating responses to the inputs that the Input Generator produces. This generation is achieved through an example solution, an oracle. To receive the output from the oracle, the Output Generator has to execute the oracle and feed it the acquired inputs. This process can be achieved by various means, but I recommend using containerization in the particular Docker, as this method provides excellent options for both isolation and scalability.

Lastly, the Main Generator should be responsible for orchestrating the Input and Output Generators. It should feed the incoming data to the Input Generator and pass the generated inputs to the Output Generator. The Main Generator must undertake the supplementary duty of overseeing the error-handling mechanism for the entirety of the project, as it is of utmost importance. This practice guarantees the immediate and efficient resolution of any problems or unanticipated malfunctions, thus minimizing any potential interruptions to the project timing. Also, the Main Generator provides an excellent opportunity to handle the necessary database manipulations, like saving the generated test cases, if needed.

Operational Mechanics

The run process:

1. The incoming data that defines the requirements for the input should be parsed.
2. Based on the parsed data, create the categories for the input generation.
3. Generate inputs.
4. Generate outputs using the oracle.
5. Format and save the generated test cases.

Establishing an efficient error-handling mechanism is crucial in order to manage a consistent execution of test case generation. Furthermore, the system should be able to offer productive feedback in the event of an unforeseen halt. Implementing the proposed solution makes it possible to guarantee prompt resolution of any challenges that may arise during the test generation process, leading to optimal productivity and minimal periods of inactivity.

Validation and Verification Principles

As the output generation should use an oracle or example solution provided by an educator, the validity of the generated outputs should only be impacted by the correctness of the educator's solution, which can be trusted.

Applying statistical techniques offers a feasible approach to examining the distribution of test inputs generated by the system. This analysis requires an in-depth review of potential gaps or biases in the propagation of the generated test inputs. Recognizing and manually verifying particular edge cases might be vital to guarantee the Generator's complete ability to handle all possible scenarios. This approach makes it possible to quickly and effectively identify and address any potential issues or limitations. In addition, displaying a high degree of diligence and comprehensiveness can enhance the effectiveness of the Generator, thus allowing the delivery of precise outcomes consistently.

It is imperative to verify the consistency of the evaluation process, which I inserted the test case generator. Through careful review of the existing test cases and the generated ones, together with an in-depth analysis of the evaluation results, it is possible to verify the consistency of the employed evaluator. This complete assessment procedure serves a crucial role in ensuring the precision and dependability of the evaluator, thus allowing for informed decision-making based on the results. Furthermore, through the in-depth observation of discrepancies, I can uphold the authenticity of my study.

In the context of generated data and its evaluation, it is crucial to consider the potential for continuous validation. Nevertheless, it is noteworthy that continuous usage can also provide this capability with limited supervision. By closely monitoring the data output and maintaining consistency in the evaluation process, it is possible to attain a significant degree of precision and dependability without the necessity for continuous validation. This methodology facilitates a more optimized and effective operational process while upholding the requisite quality assurance standard.

3.2 Research Design

The quantitive research methodology was used to assess the test case generations' effectiveness in an academic setting, mainly to provide a comprehensive evaluation

of the process. Using the quantitive method, it is more manageable to showcase both potential shortcomings and benefits of test case generation compared to the more conventional manual test creation. In addition, the goal is to provide academic institutes alongside educators with accurate results to make informed decisions for improvements in their educational methodology. With the employment of the quantitive research technique, all of the above-stated goals became more reasonable, as well as it enabled the systematic, repeatable, and accurate assessment and collection of data for the study.

The primary approach within the quantitative research methodology that I used in this thesis is the quasi-experimental approach, as it fits the design of the study exceptionally. This technique is similar to a true experiment as it also aims to demonstrate the cause-and-effect relationship between the dependent and independent variables. However, this method only requires some factors to be randomly assigned, such as eligibility threshold scores or other criteria, as it is not practical and, in some cases, not feasible. For example, I did not use random selection for the data gathering as the goal was to find a suitable environment where the experiment could be applied practically. This added leniency is the reason which makes the quasi-experiment a distinct and viable methodology in the scientific research field and sets it apart from other techniques. By employing the quasi-experiment method, researchers can assess the effects of their treatments or interventions without employing a more complex and impractically completely randomized experiment. Although this technique is less meticulously controlled than a conventional experiment, it can still yield valuable insights and contribute to our understanding of scientific phenomena.

By utilizing the above-mentioned quasi-experiment methodology's unique premise, I selected a particular category of assigned tasks to generate test cases, thereby implementing its primary benefit. This selection offers a sizeable amount of flexibility with the complexity of the assignments yet still retains the diversity of provided solutions. Moreover, applying the technique takes students' different learning styles and programming skills into account while still keeping their unique qualities and viewpoints. By limiting the kind of tasks to use for generating test cases, I can provide a more practical and easy-to-use approach for automatic test case generation along with a framework to build up to confront more complex assignments in future works. Also, by applying this kind of constraint, I can measure the potential impact on both automatic test case generation and the evaluation process closer to

reality and provide a well-researched, impartial, and precise assessment.

As previously indicated, I divided the data into two separate categories to carry out the investigation. The initial control group comprised academic assignments completed by students whose solutions had already undergone evaluation and received a grade. The following group was comprised of the same tasks, albeit with the intervention of the automated test case generator to create test cases to assess them. Subsequent sections will delve into additional information pertaining to both the data and the Generator.

In the end, quantitive research methodology, particularly a quasi-experimental approach, is a reliable way of measuring an intervention, like automatic test case generation, without the complications of complete randomization. However, it is imperative to consider its shortcomings. For example, the absence of random assignment can introduce bias. In addition, our sample's unique characteristics, like the fact that the students are new to programming or the complexity of the assignments, may limit our findings' generalizability. Finally, the Test Case Generators' complexity and test generation methods can cause a high degree of scatter, and more complex generation techniques may give results widely different from rudimentary ones.

3.3 Data Collection

3.3.1 Participants

The data used in the study was mainly made up of homework assignments submitted by first-year students currently enrolling in the Computer Science BSc—major in the Faculty of Informatics at Eötvös Loránd University. The student groups were chosen specifically because of their recent enrollment in higher education, thus leading to selecting a seminar, not demanding complex programming knowledge. Moreover, a sizeable portion of the selected students are new to programming, and automatic assignment evaluation, providing an unbiased approach to the system with a fresh outlook on coding.

The participants were selected based on their enrollment in the Object-Oriented programming course offered during the Spring semester of 2023. Only students who handed in at least one solution to a task were selected in this study, so solutions of

individuals actively engaged in the course are used in the analysis to avoid infecting the dataset of failed solutions with false positives. The objective of utilizing this criterion was to procure a more precise depiction of the student population and their levels of involvement.

The Task Management System (TMS) was utilized in the Object-Oriented Programming course for students to submit their work, as well as for me to gather data. Notably, the assignments submitted by students who maintained their enrollment in the course were considered for analysis was used. The only deviations from this pattern were observed among pupils who either failed to present a solution or opted to discontinue their enrollment at the University.

As the students whose solutions were selected to be included as data for the research are new to programming, it is expected that their usage of an automatic evaluator might cause artifacts in the data under scrutiny, such as a solution that is functionally sound but fails to produce the expected output format.

3.3.2 Data

The assignments selected for this study were coding assignments made for the Object-Oriented programming course. Object-Oriented programming is one of the earliest programming seminars that a student can attend in ELTE FI; it also uses TMS for the automatic student assignment evaluation and straightforward programming exercises, so it was a natural choice for data gathering. I decided to select solutions from the first three tasks assigned to students because these tasks are easily parameterized and are not complex in input dependencies. Furthermore, I aimed to select assignments that are straightforward programming problems for the creation of my dataset, as for such tasks, even more fundamental test case generation techniques can be used. This approach would facilitate testing more intricate assignments and the application of advanced test generation techniques in further studies. In addition, because the research method I use in this thesis requires a control group that is made from the previously mentioned assignments and the actual experiment group that was created by using the control group's solutions but the generated test cases, a direct comparison can be made to observe the impact of automatic test case generation.

Data collection from evaluations was performed utilizing TMS, a task manage-

ment and assignment evaluation solution developed by ELTE FI, and SQL queries. The utilization of this platform has ensured a consistent and reliable evaluation process by eliminating the potential for human error or bias alongside providing a straightforward procedure for end-to-end data gathering. Additionally, implementing SQL queries to directly access and group data from the underlying database has accelerated the process. Furthermore, TMS incorporates an evaluation mechanism that employs containerization technology to assess student submissions. This containerization eliminates the need for additional measures to ensure that all test cases are executed under identical conditions for each submission.

3.4 Data Analysis

Data Preparation

The raw data I received from TMS was cleared by removing incompatible solutions. Then I formatted the data so I could import it into my local instance of TMS that is set up for automatic test generation. After finishing the setup phase, I created the specification for the tasks from where the automated test generator creates the test cases. In cases where test results were missing, these were flagged and excluded from the primary analysis to avoid skewing the results.

3.4.1 Analytical Techniques

For the data analysis, I primarily used paired t-testing, which is a statistical method that compares the scores of two groups, the control group that used manually created test cases and the test group that employed automatically generated test cases. In my case, whether a solution passes or fails created the scores for the paired t-test. By utilizing this technique, I determined if there was a statistically significant difference between the two groups.

3.5 Ethical Considerations

I have effectively communicated the details of the study to the educators, who expressed their readiness to take part by submitting their sample codes and providing the details of their assignments. The active participation of the parties played

a crucial role in advancing the research objectives. Furthermore, students have already submitted their assignments to ELTE FI's TMS, making the necessary data accessible.

To ensure the protection of students' privacy, a de-identification procedure was implemented prior to receiving the data. A technique was used wherein personal identifiers, including but not limited to names and other sensitive information, were substituted with distinct codes. The code-to-identifier key was securely preserved to ensure confidentiality. Implementing this measure guaranteed that solely authorized members of the research team were granted access to the data, thus preserving the confidentiality and integrity of the information.

In order to guarantee ideal data security, all information was stored on a specifically designated research computer. Also, to mitigate the risk of potential data loss, periodic backups were performed to ensure the preservation of critical information and prevent any compromise or loss.

During the study, I considered the potential consequences of the findings on the students. An example of this would be if the study revealed that the automated test case generation process had a significant impact on the scores of assignments; this could potentially lead to changes in the way that grading practices are carried out. Measures were taken to ensure that the study did not unfairly advantage or disadvantage any students.

Chapter 4

Implementation

4.1 Task Management System - TMS

The Task Management System, or TMS (`https://tms-elte.gitlab.io/`), is a highly efficient student assignment evaluator and grader system developed explicitly by the Eötvös Loránd University Faculty of Informatics. We designed this system to assist and streamline student assignments' evaluation and grading process, ensuring they are assessed and graded promptly and accurately. With its advanced features and user-friendly interface, TMS is an invaluable tool for educators and students. It is a reliable and efficient means of managing assignments and ensuring academic success. TMS is a complex system that encompasses a variety of possible tasks that a student can accomplish. In addition, it provides opportunities for teachers to manage student assignments and exams. TMS also offers various ways for plagiarism checking concerning student assignments. However, in my case, the most essential part is the assignment evaluator and the process a teacher must go through to use it.

4.1.1 Assignment evaluation process

After an instructor logs into the system, they will first encounter the group selection page, where they can choose which study group they wish to use for work as shown in Figure 4.1. Only groups where the instructor was assigned to teach, or help are visible.

Figure 4.1: TMS group list

After they select a group, they will see the various details of the group, like its id, name, and the course code that the group belongs. Here the instructor can edit, delete, or duplicate the course if needed—four tabs on the group page show relevant information to the instructor. The assignments tab, the default tab, shows all the tasks created for the group as displayed in Figure 4.2; here, the instructor can also set up new ones. All students assigned to the group are listed on the students tab, and new students can be added. Also, teachers can add personalized notes for students here. The instructors tab is also similar to the students tab, as all instructors are listed here, and new ones can also be assigned. Lastly, in the statistics tab, instructors can view various performance indicators in the group, like the points scored or assignment handling time.

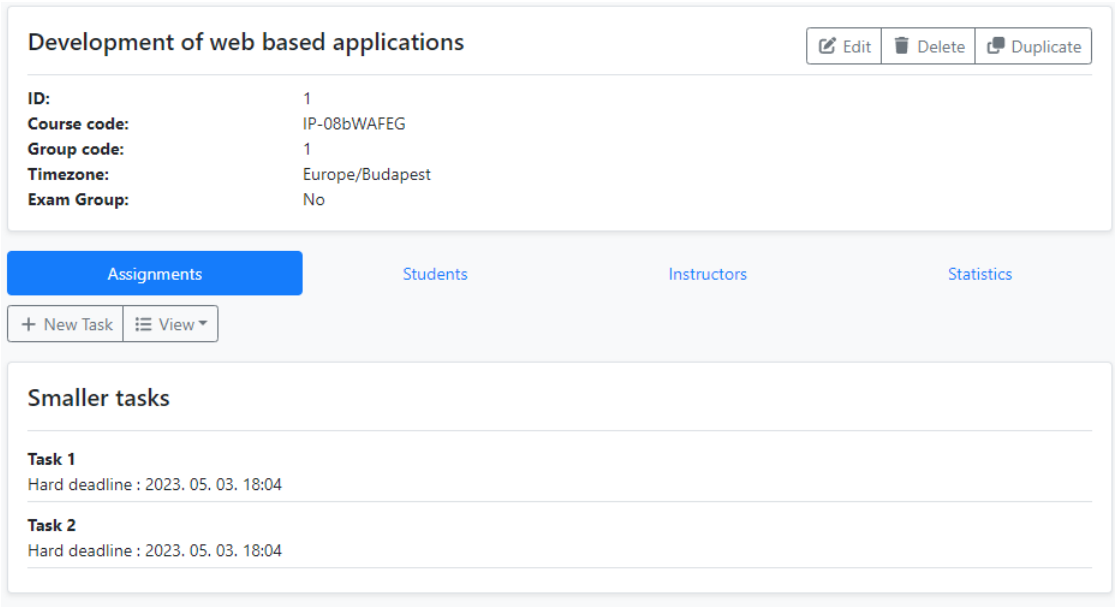

Figure 4.2: TMS group page assignment tab

After selecting a task to work on, the instructor will be presented with the task

page, where they can edit or delete the task and view pertinent information as shown id Figure 4.3. In addition, four tabs show different groupings of information and interactions that the educator can select. The default is the solutions tab; the teacher can view submitted student solutions here. On an individual student submission, an instructor can view various information, like but not limited to the upload time, the status of submission that includes the automatic evaluation and the manual acceptance, the grade, instructor notes on the submission, and the results of static code analysis. The educator can also download or grade a submission in this section.

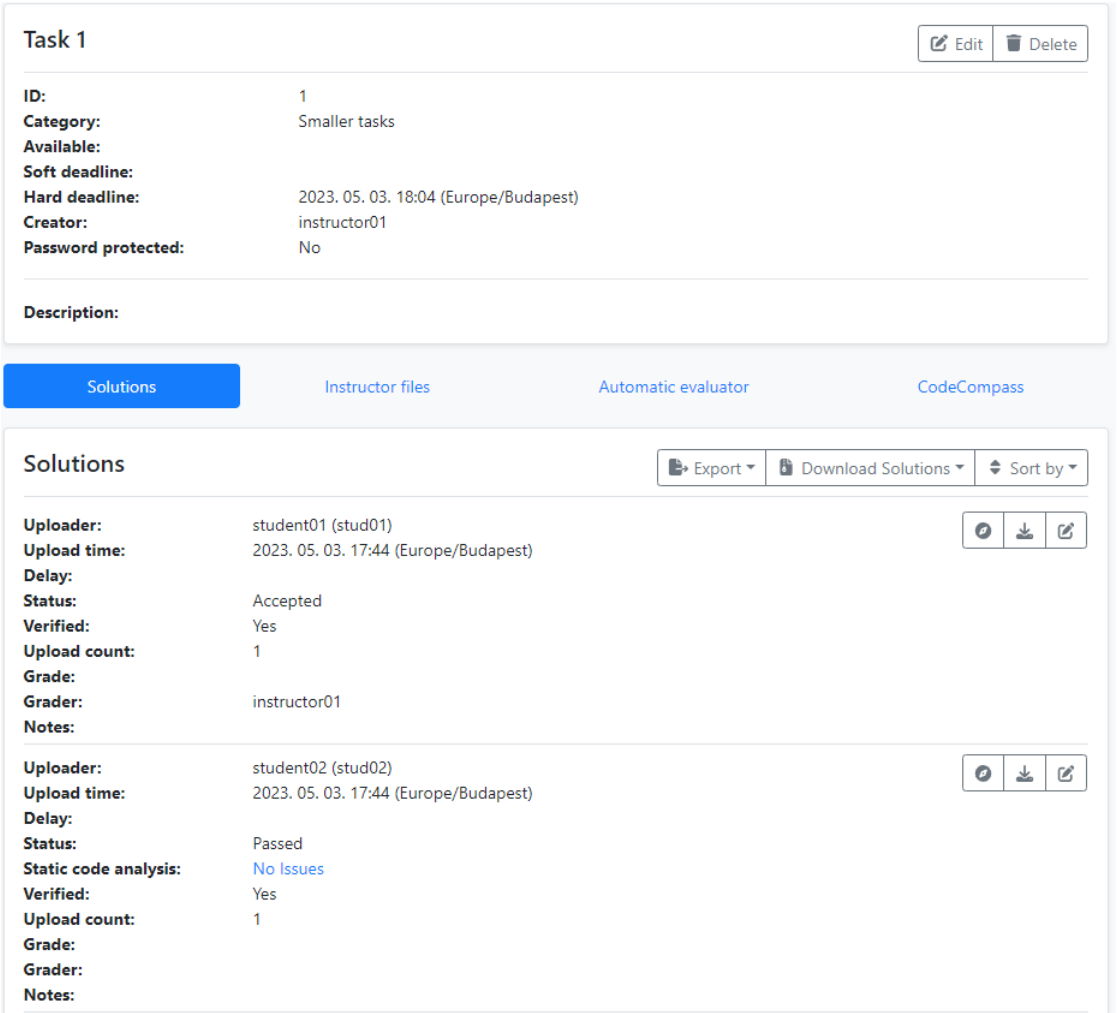

Figure 4.3: TMS task page solutions tab

Educators can upload relevant and suitable files for their students within the instructor files tab. This feature allows for a streamlined and organized approach to sharing important information with students and assessing their progress.

Teachers can create scripts for package installation and compilation of instructions used by the CodeCompas system that is integrated with TMS in the

CodeCompas tab. In addition, CodeCompas offers helpful coding tools, such as a code analyzer and viewer, that support effective coding techniques.

The tutors encounter three sections responsible for different settings in the automatic evaluator tab. The first is the environment section, where the automatic evaluator's environment can be adjusted as displayed in Figure 4.4. TMS uses containerization for the testing of student applications, in particular docker, so a docker image or a docker file is mandatory to set the suitable environment. Also, instructor files can be added here if the submission evaluation needs it. The second is the Automatic tester section; the educators can set the compile and run instructions for student submissions. This section is also where the instructor can set the manual test cases or set up the automatic test case generation. Finally, in the static code analysis section, the teachers can set the necessary settings for the static code analyzer, like analyzer tool instructions or source files, to ignore. TMS also provides templates that fill all three of the section with predefined settings.

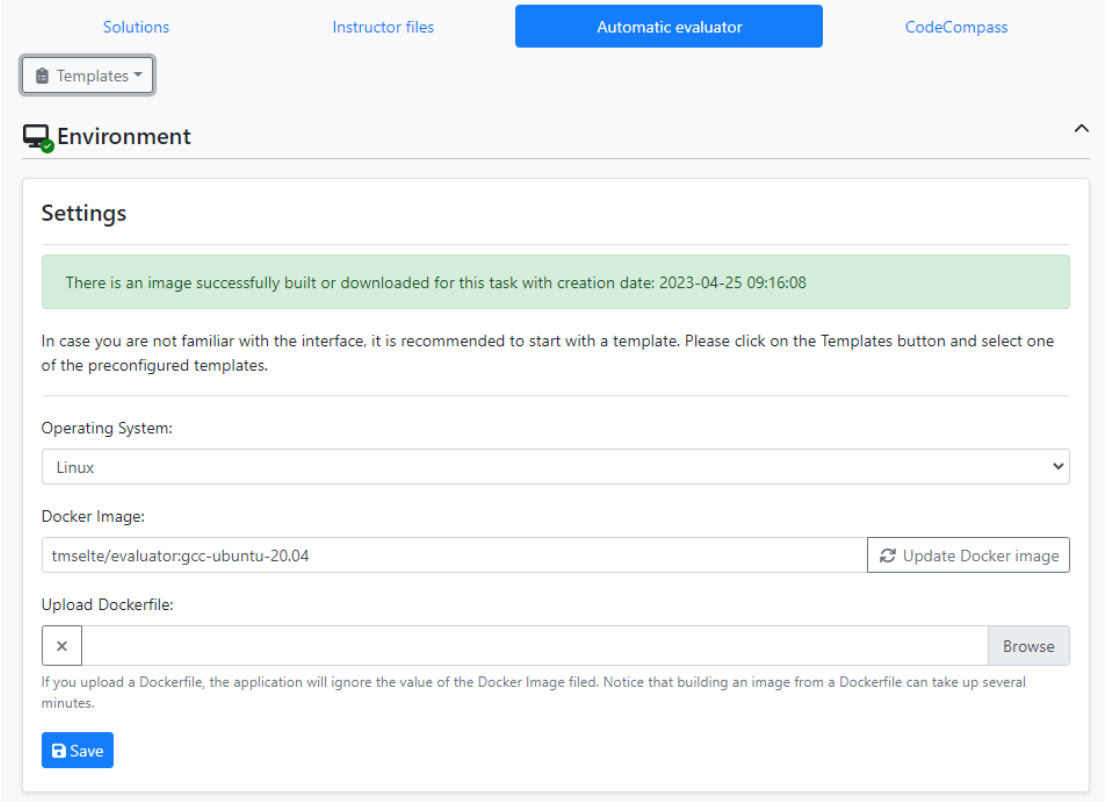

Figure 4.4: TMS task page automatic evaluator tab

Following the guidelines above, an instructor can set up all the necessary settings so their task can be automatically tested. Figure 4.5 displays the workflow for the entire process.

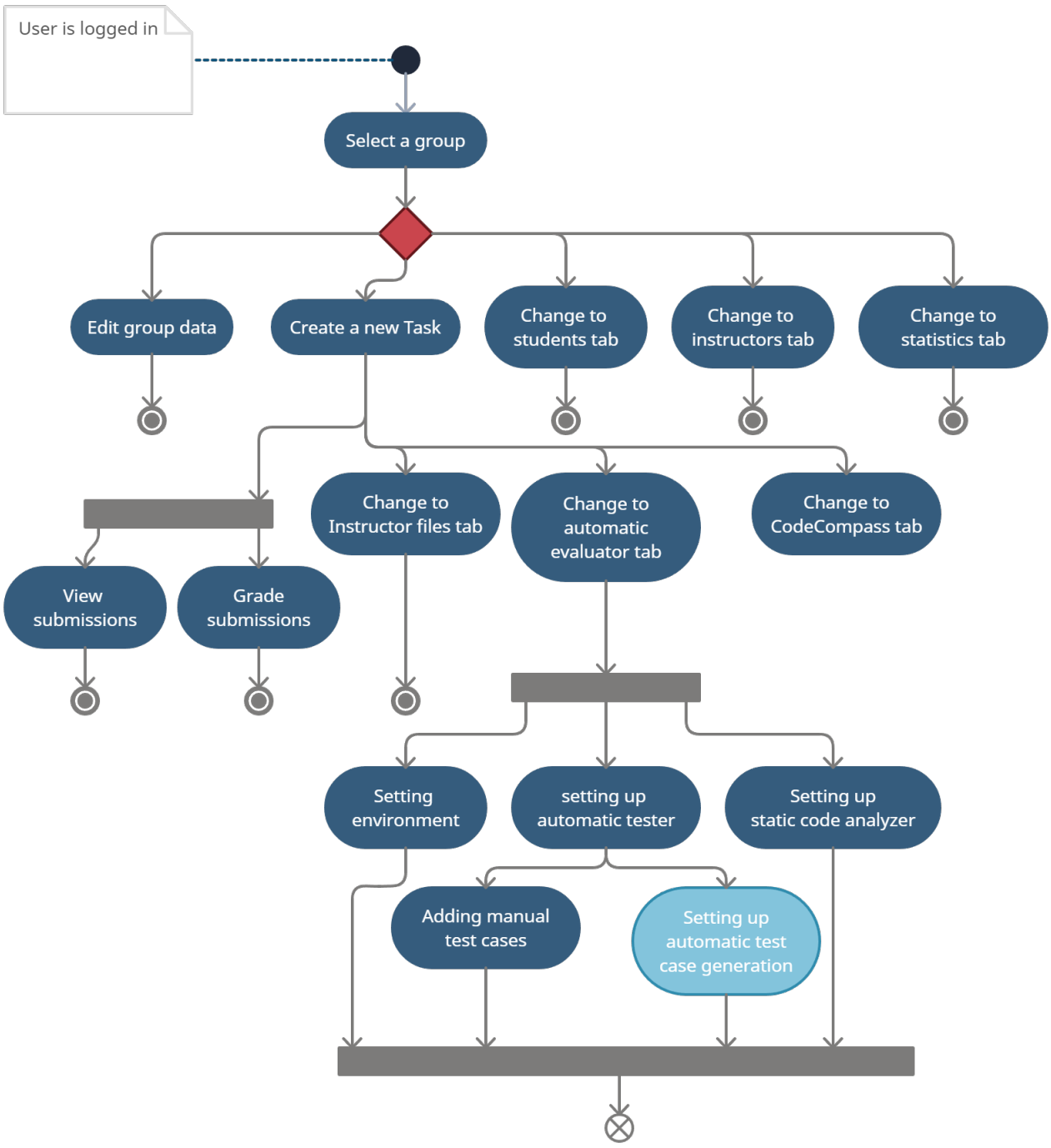

Figure 4.5: Workflow in TMS for an instructor to setup automatic testing

4.1.2 TMS technical overview

TMS is a complex system that follows a server-client paradigm mainly because its backend pursues the REST architecture. The Representational State Transfer (REST) is an architectural style that was conceptualized by Roy Fielding in the year 2000. It aims to provide a framework of engineering principles and interaction constraints for developing distributed information systems. RESTful systems conform to six constraints: client-server constraint, stateless constraint, cache constraint, layered system constraint, code-on-demand constraint, and uniform interface constraint. From these six rules, code-on-demand is optional. Using these rules makes it more manageable to build a system that is spread out. However, deviating from these limitations may result in further complications in the system's architecture[32].

The backend is written in PHP, utilizing the Yii2 framework. The Yii framework is a PHP-based, component-oriented platform designed to facilitate the swift creation of contemporary web applications with superior performance. PHP is a suitable programming language for developing various web applications, particularly large-scale ones, such as portals, forums, content management systems (CMS), e-commerce projects, RESTful Web services, and other similar applications[33]. Yii is a comprehensive full-stack framework that offers a plethora of pre-built and tested features, including query builders and ActiveRecord for relational and NoSQL databases, support for developing RESTful APIs, multi-tier caching support, and other functionalities. The framework adopts the Model-View-Controller (MVC) architectural paradigm and advocates for the code arrangement per this pattern. Yii also exhibits high extensibility, allowing developers to modify or substitute virtually all components of the framework's core code. In addition, it boasts a strong core development team and a sizable group of professionals who consistently contribute to the platform's development[33].

The frontend is written in React TypeScript, a hybrid of React and TypeScript, two formidable technologies. React is a JavaScript library utilized for constructing user interfaces, whereas TypeScript is a statically typed extension of JavaScript. TypeScript is recognized for its robust type-checking capability, which aids in developing highly resilient and scalable applications. The integration of TypeScript with React development yields beneficial outcomes for developers.

4.2 Backend implementation

4.2.1 Database

Regarding the database, TMS takes a code-first approach, indicating that the code will construct the schema automatically rather than by hand. This strategy guarantees correctness and consistency in the database schema while saving time and effort. Furthermore, with TMS, the development process is more simplified and efficient since developers can now concentrate more on the code itself and less on the database structure.

I only needed to create one new table in the database with relevant information regarding the test case generation as displayed in Figure 4.6. The most important fields are the 'fieldCount' that represents the number of input fields, the 'generationCount' that holds the required number of test cases to be generated, and the 'data' filed, which is responsible for storing the actual data required for the generation. The 'data' fields structure is as follows:

- It mainly consists of an associative list, particularly a collection of objects, where the object's name is a number.
- Inside a primary object, one common property named 'type' will help decide what kind I can read further.
- After the 'type' property, there can be four other properties:
  'length,' if the type is a string and defines the length of the string to be generated, 'enumValues,' in case the type is an enum, that is a list of strings representing an enum member. If the type is an integer, there are 'upperBounds' and 'lowerBounds,' which are responsible for a bounding whom the integer should be generated between.

A foreign key was added to the newly created table to connect to the task table as it is the binding block that contains most data necessary to run the evaluator, like the name of the docker container. Also, a new field was added to the test cases table to differentiate generated and manual test cases.

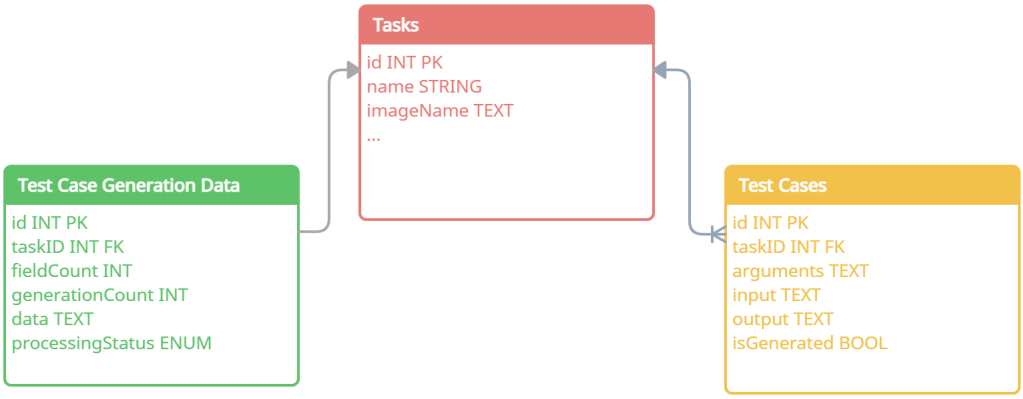

Figure 4.6: Changed database entries

As TMS uses a code-based approach, I also created a Model class to represent the test generation data. In this class, alongside the necessary fields, validation rules and scenarios were defined to manage correct and straightforward data flow.

4.2.2 API controller

I created three new endpoints as TMS's backend uses REST API to publish its functionality to retrieve the information necessary for test case generation as shown in Figure 4.7.

- I created a GET endpoint so an end-user can retrieve an already existing record, or in the case of no such record, a new empty record can be found that is flagged as 'Not Configured.'
- The POST endpoint creates an entirely new data record and requires the end-user to upload an example solution alongside the other fields.
- Lastly, the PUT endpoint updates existing entries; no example solution is necessary for this call.

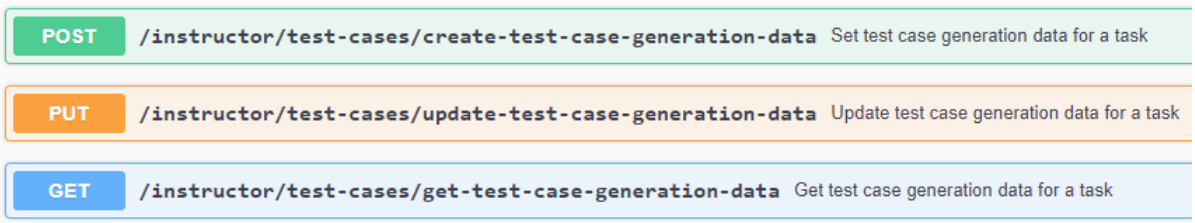

Figure 4.7: TMS endpoints for test case generation

Storing the uploaded example file in a different folder structure than directly in the database is essential. Also, to separate the infrastructure from the representation layer, I wrapped the Model class into a Resource class like a DTO.

4.2.3 CLI controller

During the design process of the implementation, I concluded that to prevent ad-hoc high load on the TMS server, it is not practical to start the test case generation right after the end-user uploaded the data necessary for a generation. For example, if an educator starts to create a new data for generation, and if an exam is in progress, it could cause problems with the queuing of the evaluations and slow down the system; this can be mitigated with rules, however generally, it is not expected for student solutions to be uploaded right after a task was created, as this is the optimal time to create test cases too, so the generation can be delegated to a cron job that runs during the downtime of TMS, and for this purpose, I created a CLI controller to run.

The controller is responsible for selecting the eligible test case generation data entries from the database. These entries are then passed down to the generator. The CLI controller also saves the status of the generation run and the created entries.

4.2.4 Generator

Upon thorough deliberation, I have employed a pure random testing methodology for the generator. The approach mentioned above facilitates the prompt generation of test cases, which is crucial for accomplishing our task. Furthermore, implementing this doctrine makes it possible to guarantee that the testing procedure is efficient and effective, ending in a more streamlined and successful result. The Test Generator consists of three principal parts: the Input Generator, the Output Generator, and the Test Generator itself.

The Input Generator has two main tasks to accomplish, the first is to parse the incoming data in the form of a JSON string, and the second is to generate data based on the parsed input.

- In the case of ENUM inputs, the data generation process is simple; the ENUM elements parsed from the input as a list are rearranged and then selected to fill the output list until the specified numerosity is achieved.

- For the type String, a randomly generated base64-based string is generated with the given length for the input. Then I repeat this process until the number of needed inputs is reached.

- For the Integer type, firstly, the interval between the upper and lower bounds from the input is spliced into equivalent partitions based on the necessary number of test cases to generate. Then, a random number is selected as an input value in each section.

In the Test Generator, firstly, I call the Input Generator. Then a list of TestCase objects is created, and the generated inputs from the Input Generator are merged and placed into their inputs field. Finally, with the test case inputs in place, the Output Generator is called to generate outputs to the inputs and finalize the generation process.

The Output Generator operates with a list of test cases where the inputs field is filled and the example solution that an educator uploaded as an oracle. Firstly all the necessary files to run the example solution, like compile instructions, are copied into a temporary folder. Then a new docker container is created to run the example solution. Next, all of the prepared files are transported into the docker container. After compilation, I feed the list of inputs into the docker image as a standard input and wait for the output of the provided example solution. Finally, return the now completed test case list.

4.3 Frontend implementation

I wrote the frontend in React TypeScript to fit in with the implementation of TMS. The primary consideration regarding the frontend was to create a user-friendly and intuitive solution as displayed in Figure 4.8. Aside from the API calls, the modification in the frontend consists of the dynamic form so that the end user can provide the necessary information for the test case generation. The form starts with two number inputs to give, from which the number of input arguments is the more crucial from an implementation standpoint, as new form elements are created depending on the input on this field. These newly generated dropdowns then determine the sub-fields depending on their selection. The difficulties of this design come from the dynamic nature of the form, updating and transforming the

data, and binding the underlying class to the form. I remedied these ordeals with the help of 'useState' and to capture the fields that incite changes in the form 'useEffect' to populate the form if data is available from the backend. I solved the binding to the form with a class that is a superset of the sub-fields instead of the specific class for all variations of sub-fields.

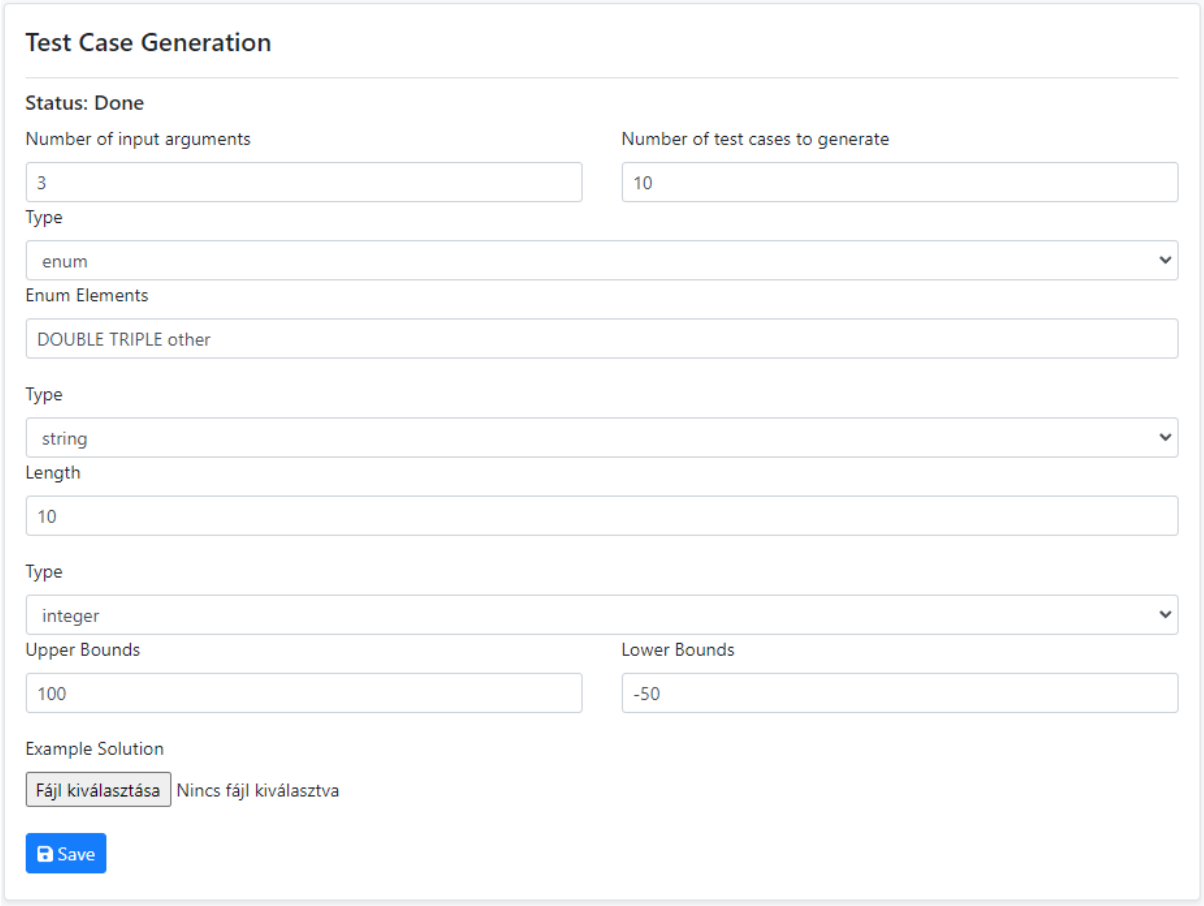

Figure 4.8: Test case generation form

Chapter 5

Results

In this study, I aimed to determine the effects of automatic test case generation on the evaluation of student submissions compared to manually created or completely manual test cases. For this reason, I was provided 183 thoroughly anonymized student submissions grouped by the four tasks to which they were submitted, as discussed in Section 3.3. I utilized my implementation of an automatic test generator built into TMS to generate test cases and run them, then compared them to the evaluation results of the already run manual test cases. In the following sections, I will in-depth answer the three main research questions.

5.1 Evaluation results

The assignments evaluated with the manually created test cases passed 88% of the time in the received dataset. In comparison, the student solutions tested with the automatically generated test cases received a passed score 82% of the time, as seen in Figure 5.1. Therefore, we can conclude that the generated test cases detected 6% more failures in this case. Albeit not being the significant percentage expected, it still shows that manual test cases can be replaced by automatic test cases and improve the evaluation process.

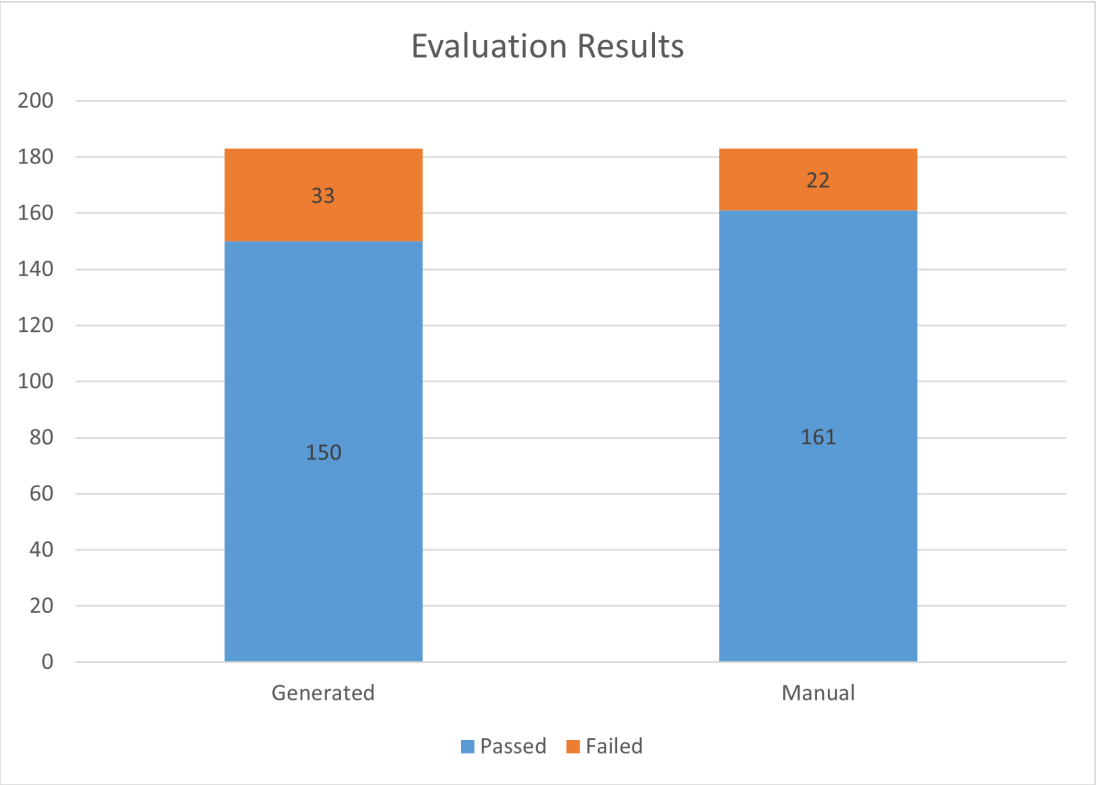

Figure 5.1: Automatic evaluation results

In Figure 5.2, I show the distribution of the evaluation results from the generated test cases. Again, it is visible that the majority of the errors come from programming errors; other minor errors occur due to the lack of attention on the student's part.

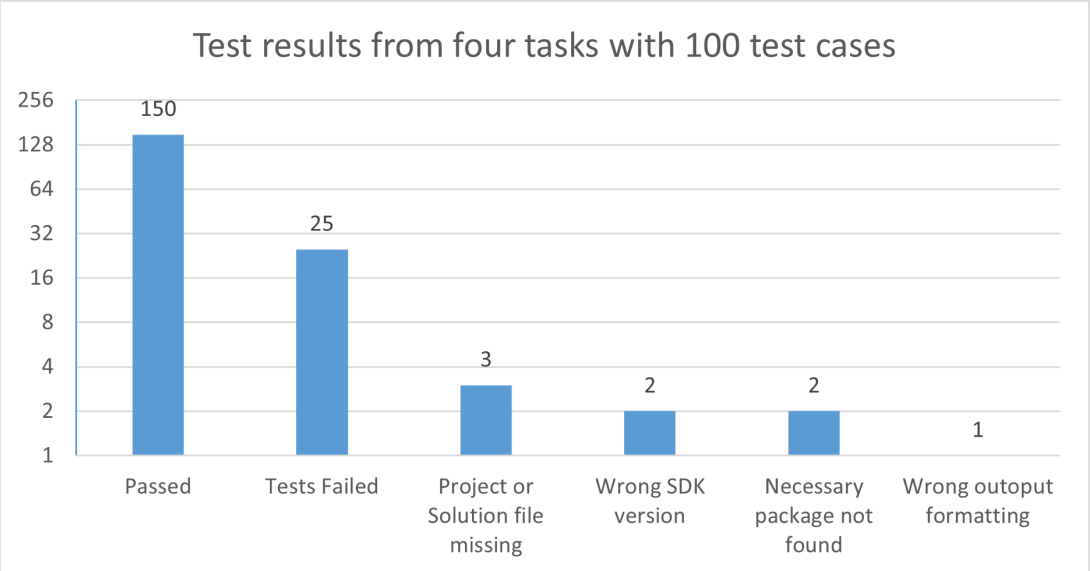

Figure 5.2: Automatic evaluation results for generated test cases on four tasks using 100 test cases

5.2 Comparison of generated and manually created test cases

Does automated test case generation change the evaluation results of assignments?

It is visible in Figure 5.3 that the generated test cases find most of the errors that are detected by manual test cases, and in some cases, it detects more errors, namely 37% more than in the control group. It also finds that in the case of this dataset, all of the manually viewed solutions with errors. Nevertheless, it is noteworthy that 5% of failures were not detected by the automatically generated test cases, in contrast to the manually created ones. This detection oversight can be mitigated with the increase in the number of test cases, as this would give a higher percentage of coverage of the assignment under test or with the introduction of more complex testing algorithms. So we can conclude that the automatic test case generation can benefit the evaluation of student assignments, but not as a 100% replacement for educator oversite.

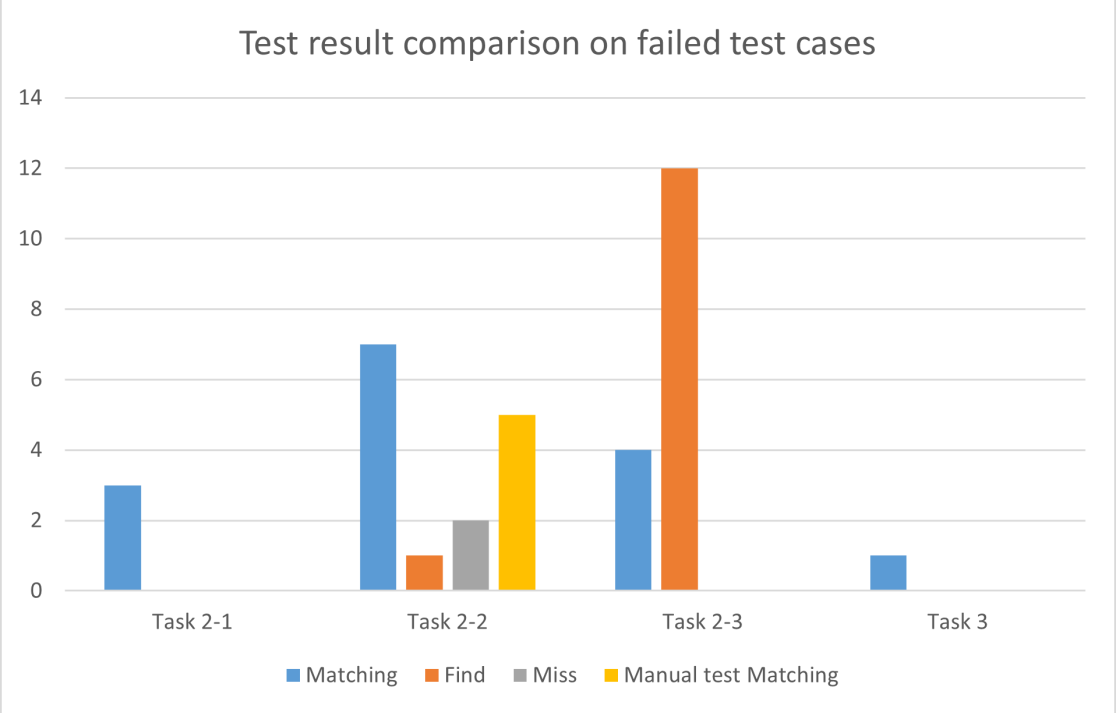

Figure 5.3: Comparison of test results on failed test cases between generated and manually created/completely manual

5.3 Test case generation time

In Figure 5.4, it can be seen that the generation of test cases takes little time, as 100 generated test cases can be created in about a minute. It is also noteworthy that time increases linearly with the number of test cases to be generated. A minor deviation can be observed depending on the number of inputs and their complexity, but it is marginal. The generation is a fast and fully automated process that can be made a timed job. The usage time of the test case generator can vary widely depending on the type of inputs the user wants to generate, but the number of inputs has the most impact.

Does automated test case generation improve the evaluation process for educators?

During my testing, the simplest tasks took about one to three minutes to set up, while the more complex ones took about 5 minutes. It is safe to say that the introduction of automatic test case generation can save time on an educator's part. Nevertheless, it is necessary to conclude further, more diverse studies to solidify my findings.

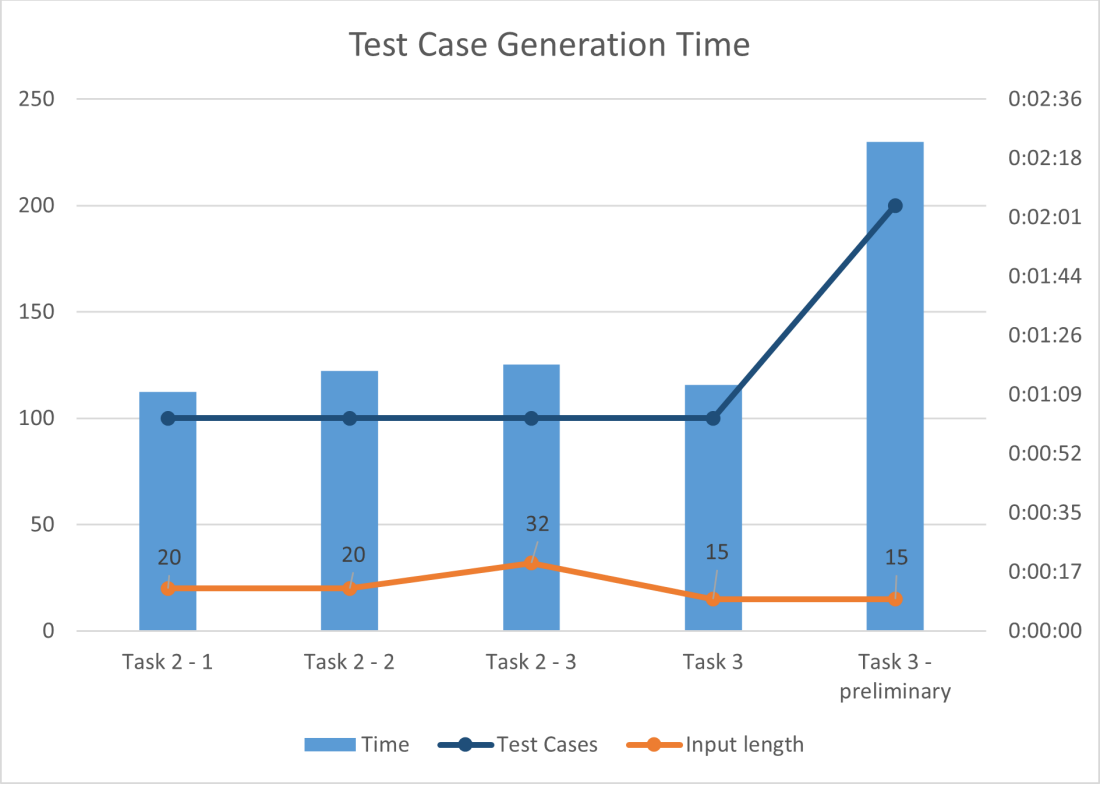

Figure 5.4: Test case generation time

5.4 Edge case detection

Can automated test case generation find edge cases like manual tests?

While analyzing the evaluation results, I noticed a pattern among the failed assignments. In the same tasks, most test cases that caused the student submissions to fail were the same. This discovery points towards a conclusion that these kinds of test cases are edge cases for the given task, and as a side effect of the evaluation, these edge cases can be found. Although the possibility of these test cases being edge cases is high, it is worth noting that without a requirement analysis, it is hard to conclude it with 100% certainty. Also, even if the found test cases are edge cases, the evaluation of automatic test cases cannot guarantee the discovery of all possible edge cases, as it is not designed for that purpose. To conclude, evaluating the generated test cases has a high likelihood of finding edge cases if the same test cases cause the failure of solutions, but it cannot be concluded without the analysis of the given task.

Chapter 6

Conclusion

Automatic test case generation enables various possibilities for the improvement of educational processes. This is an enormous topic for research, with many possible paths to choose and improve on. During my research, I dwelled deep into various testing techniques to explore their potential cases in test generation.

After thorough research conducted on 183 student solutions, I can conclude that the introduction of automatic test case generation has tangible benefits. Aside from the improvements in future discovery during automated evaluation, it also provides noticeable gains in regard to the time it takes for educators to create test cases. Furthermore, it has the potential to detect edge cases for the tasks that the solutions are solving.

The test case generation achieves this while performing exceptionally well regarding runtime. Moreover, as a notable correlation between the runtime and the number of generated test cases could only be observed, it is also effortlessly scalable.

However, it is essential to note that a rudimentary testing technique like pure random testing, which was used in my study, is not always able to replace manual oversight of grading. As it was shown in Chapter 5, it cannot necessarily find all failures that a more precise, manually created test case can filter out. Moreover, edge case detection is, albeit likely; this technique cannot guarantee to find all test

cases, as well as it cannot ensure that the found edge cases are real ones.

Future work

A sizeable amount of possible future research can be made regarding the effects of automatic test case generation on the evaluation of student assignments. Starting with investigating different testing techniques' effects on test case generation and finding the optimum method to find the highest amount of errors while still cutting down on the time an educator needs to spend on test case creation.

It is also important to consider expanding the possible types of inputs the test case generation framework can handle, like adding dependent types or data structures. Furthermore, developing the test framework to work with more complex tasks is worth looking into.

Lastly, comprehensive live testing of the automatic test case generation is imperative to discover any unexpected behavior that could not appear during a quasi-experiment. Moreover, in this setting, it is also possible to collect accurate data on the time-saving aspect of the study to cement my findings further.

Acknowledgements

I want to express my sincere thanks to my supervisor Máté Cserép for his great help and oversight in regard to my thesis. My fellow students who helped with the laboratory work for TMS, especially Péter Kaszab, who helped me understand the inner workings of TMS, and Levente Jakab, who provided support regarding some miscellaneous laboratory tasks. Lastly, I would like to thank all of the other colleagues, classmates, and my family that cheered me on during the creation of this thesis.

Bibliography

[1] Tracy Camp et al. "Generation CS: The Growth of Computer Science". In: *ACM Inroads* 8.2 (May 2017), pp. 44–50. ISSN: 2153-2184. DOI: 10.1145/3084362. URL: https://doi.org/10.1145/3084362.

[2] Hussam Aldriye, Asma AlKhalaf, and Muath Alkhalaf. "Automated Grading Systems for Programming Assignments: A Literature Review". In: *International Journal of Advanced Computer Science and Applications* (2019).

[3] Yuki Akahane, Hiroki Kitaya, and Ushio Inoue. "Design and evaluation of automated scoring Java programming assignments". In: June 2015, pp. 1–6. DOI: 10.1109/SNPD.2015.7176255.

[4] Stephen H. Edwards and Manuel A. Perez-Quinones. "Web-CAT: Automatically Grading Programming Assignments". In: *SIGCSE Bull.* 40.3 (May 2008), p. 328. ISSN: 0097-8418. DOI: 10.1145/1597849.1384371. URL: https://doi.org/10.1145/1597849.1384371.

[5] Adidah Lajis et al. "A Review of Techniques in Automatic Programming Assessment for Practical Skill Test". In: *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 10.2-5 (July 2018), pp. 109–113. URL: https://jtec.utem.edu.my/jtec/article/view/4394.

[6] Surendra Gupta and Shiv Dubey. "Automatic Assessment of Programming assignment". In: *Computer Science & Engineering: An International Journal* 2 (Feb. 2012). DOI: 10.5121/cseij.2012.2107.

[7] P.J. Schroeder, P. Faherty, and B. Korel. "Generating expected results for automated black-box testing". In: *Proceedings 17th IEEE International Conference on Automated Software Engineering,* 2002, pp. 139–148. DOI: 10.1109/ASE.2002.1115005.

[8] Francesca Saglietti, Norbert Oster, and Florin Pinte. "White and grey-box verification and validation approaches for safety- and security-critical software systems". In: *Information Security Technical Report* 13 (Dec. 2008), pp. 10–16. DOI: 10.1016/j.istr.2008.03.002.

[9] S. Nidhra. "Black Box and White Box Testing Techniques - A Literature Review". In: *International Journal of Embedded Systems and Applications* 2 (June 2012), pp. 29–50. DOI: 10.5121/ijesa.2012.2204.

[10] A.S. Boujarwah and K. Saleh. "Compiler test case generation methods: a survey and assessment". In: *Information and Software Technology* 39.9 (1997), pp. 617–625. ISSN: 0950-5849. DOI: https://doi.org/10.1016/S0950-5849(97)00017-7. URL: https://www.sciencedirect.com/science/article/pii/S0950584997000177.

[11] Shaoying Liu and Yuting Chen. "A relation-based method combining functional and structural testing for test case generation". In: *Journal of Systems and Software* 81.2 (2008). Model-Based Software Testing, pp. 234–248. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2007.05.036. URL: https://www.sciencedirect.com/science/article/pii/S0164121207001276.

[12] Phyllis G. Frankl and Elaine J. Weyuker. "Testing software to detect and reduce risk". In: *Journal of Systems and Software* 53.3 (2000), pp. 275–286. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(00)00018-2. URL: https://www.sciencedirect.com/science/article/pii/S0164121200000182.

[13] Zhang Zhonglin and Mei Lingxia. "An improved method of acquiring basis path for software testing". In: *2010 5th International Conference on Computer Science & Education*. 2010, pp. 1891–1894. DOI: 10.1109/ICCSE.2010.5593820.

[14] Mohd Khan. "Different Approaches To Black box Testing Technique For Finding Errors". In: *International Journal of Software Engineering & Applications* 2 (Oct. 2011). DOI: 10.5121/ijsea.2011.2404.

[15] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. "Chapter Four - Recent Advances in Automatic Black-Box Testing". In: ed. by Atif Memon. Vol. 99.

Advances in Computers. Elsevier, 2015, pp. 157–193. DOI: https://doi.org/10.1016/bs.adcom.2015.04.002. URL: https://www.sciencedirect.com/science/article/pii/S0065245815000315.

[16] Joe W. Duran and Simeon C. Ntafos. "An Evaluation of Random Testing". In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 438–444. DOI: 10.1109/TSE.1984.5010257.

[17] Atif Memon et al. "The first decade of GUI ripping: Extensions, applications, and broader impacts". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 11–20. DOI: 10.1109/WCRE.2013.6671275.

[18] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. "Interpolated N-Grams for Model Based Testing". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 562–572. ISBN: 9781450327565. DOI: 10.1145/2568225.2568242. URL: https://doi.org/10.1145/2568225.2568242.

[19] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. "Using Binary Decision Diagrams for Combinatorial Test Design". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 254–264. ISBN: 9781450305624. DOI: 10.1145/2001420.2001451. URL: https://doi.org/10.1145/2001420.2001451.

[20] Linbin Yu et al. "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 242–251. DOI: 10.1109/ICST.2013.35.

[21] Tafline Murnane, Karl Reed, and Richard Hall. "On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis". In: *2007 Australian Software Engineering Conference (ASWEC'07)*. 2007, pp. 274–283. DOI: 10.1109/ASWEC.2007.35.

[22] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Formal Analysis of the Effectiveness and Predictability of Random Testing". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA

'10. Trento, Italy: Association for Computing Machinery, 2010, pp. 219–230. ISBN: 9781605588230. DOI: 10.1145/1831708.1831736. URL: https://doi.org/10.1145/1831708.1831736.

[23] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Random Testing: Theoretical Results and Practical Implications". In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 258–277. DOI: 10.1109/TSE.2011.121.

[24] Manuel Oriol. "Random Testing: Evaluation of a Law Describing the Number of Faults Found". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 201–210. DOI: 10.1109/ICST.2012.100.

[25] Alex Groce et al. "Swarm Testing". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, pp. 78–88. ISBN: 9781450314541. DOI: 10.1145/2338965.2336763. URL: https://doi.org/10.1145/2338965.2336763.

[26] Huai Liu et al. "Adaptive Random Testing by Exclusion through Test Profile". In: *2010 10th International Conference on Quality Software*. 2010, pp. 92–101. DOI: 10.1109/QSIC.2010.61.

[27] Johannes Mayer and Christoph Schneckenburger. "An Empirical Analysis and Comparison of Random Testing Techniques". In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. ISESE '06. Rio de Janeiro, Brazil: Association for Computing Machinery, 2006, pp. 105–114. ISBN: 1595932186. DOI: 10.1145/1159733.1159751. URL: https://doi.org/10.1145/1159733.1159751.

[28] Leila Naslavsky, Hadar Ziv, and Debra J. Richardson. "MbSRT2: Model-Based Selective Regression Testing with Traceability". In: *2010 Third International Conference on Software Testing, Verification and Validation*. 2010, pp. 89–98. DOI: 10.1109/ICST.2010.61.

[29] Daniel Sinnig, Ferhat Khendek, and Patrice Chalin. "A Formal Model for Generating Integrated Functional and User Interface Test Cases". In:

2010 Third International Conference on Software Testing, Verification and Validation. 2010, pp. 255–264. DOI: 10.1109/ICST.2010.56.

[30] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[31] Leonardo Mariani et al. "G-RankTest: Regression testing of controller applications". In: *2012 7th International Workshop on Automation of Software Test (AST)*. 2012, pp. 131–137. DOI: 10.1109/IWAST.2012.6228981.

[32] Bob Jones. *Rest architecture*. July 2020. URL: https://www.redhat.com/en/blog/rest-architecture.

[33] URL: https://www.yiiframework.com/doc/guide/2.0/en/intro-yii.

List of Figures