Simulating Student Mistakes to Evaluate the Fairness of Automated Grading

Benjamin Clegg*, Siobhán North†, Phil McMinn‡
Department of Computer Science
University of Sheffield
Sheffield, United Kingdom
Email: *bsclegg1, †s.north, ‡p.mcminn@sheffield.ac.uk

Gordon Fraser
Faculty of Computer Science and Mathematics
University of Passau
Passau, Germany
Email: gordon.fraser@uni-passau.de

***Abstract*—The use of autograding to assess programming students may lead to unfairness if an autograder is incorrectly configured. Mutation analysis offers a potential solution to this problem. By simulating student coding mistakes, an automated technique can evaluate the fairness and completeness of an autograding configuration. In this paper, we introduce a set of mutation operators to be used in such a technique, derived from a mistake classification of real student solutions for two introductory programming tasks.**

***Index Terms*—automated grading; mutation analysis; programming mistakes;**

I. Introduction

Recent years have seen an unprecedented growth in the enrollment of students in higher education Computer Science degree programs [1]. This surge of students has placed significant pressure on institutions and educators, particularly for the assessment of learning outcomes. Educators often turn to automated grading and feedback systems in order to reduce the time and resources required to perform such assessments [2].

Educators deploying autograders must configure them for each assessment, such as through the definition of test suites and static analysis tools. If a grading configuration tests for knowledge outside a task's learning objectives, students may be unfairly graded for a lack of knowledge that they have not yet been provided with. Moreover, a student will get an unfair grade if they make a minor mistake that is detected by disproportionately many similar tests. Also, inaccurate grading unfairly impacts students that are correct, failing to reinforce desirable practices. Furthermore, incomplete grading configurations prevent students' mistakes from being identified, reinforcing negative behavior, rather than correcting it. These problems are likely to occur in existing autograded programming tasks, since educators may have limited or even incorrect knowledge of mistakes that students make [3]. The complexity of configuring an autograder may exacerbate these issues. For example, when using multiple grading tools, educators must define weights for each [4].

We contend that mutation analysis can be employed to inform educators of potential inaccuracy, incompleteness, and unfairness of an autograding configuration. Mutation analysis involves making changes to a correct program based on a set of mutation operators, which have proven to be an effective means of simulating real faults in software [5]–[7]. Existing mutation operators do not entirely encapsulate the coding mistakes that students make, so new operators must be introduced. By identifying the types of mistakes that students make when writing software, we are able to define a set of mutation operators that simulate them. These simulated mistakes can be executed by a grader. Their detection reveals which mistake types an autograder misses, and those that may be unfairly punished. In conventional mutation analysis, functionality is the only focus. In education, we transform students into competent software engineers, so the style and quality of code must be considered, and such mistakes should also be simulated.

In this paper, we present two key contributions:

- A *classification* of student mistakes derived from an analysis of 126 solutions submitted by students for two introductory programming tasks, presented in Section II.
- 18 identified *mutation operators* that simulate each of these mistake classes, presented in Section III.

These mutation operators are required to realize a technique to evaluate autograding configurations and inform tutors of potential improvements, which we propose in Section IV. This technique would also weight individual components of a grading configuration (e.g., unit tests), using a metric that balances the impact of individual mistake classes. This improves fairness, as different types of mistakes will not disproportionately affect grades.

II. Initial Investigation

In order to derive appropriate mutation operators, we conducted a qualitative analysis targeting mistakes present in programs written by students for an introductory programming course. We identified not only faults that impact functionality, but also violations of style and code quality guidelines, to fully capture the coding mistakes made by students.

A. Dataset

In this study we use real solutions written by 63 students for two separate tasks in an introductory Java programming module. Both assignments require the students to use a course-specific library for handling input and output. Students were graded on functionality, style, and code quality.

TABLE I

Observed student code mistakes

Observed Mistake Class	Description	Frequency					
		Task 1		Task 2		Total	
		Count	%	Count	%	Count	%
Literal Value Repetition	Non-zero literal values repeated where constants can be defined.	55	87.3	45	71.4	100	79.4
Statement Repetition	A statement is repeated unnecessarily.	44	69.8	34	54.0	78	61.9
Poor Indentation	Misaligned indentation, or no indent after a brace.	20	31.7	31	49.2	51	40.5
Constants Defined as Variables	Constants are defined without the use of `final`.	9	14.3	31	49.2	40	31.7
Overly Long Lines	Any line of code exceeds 100 columns in width (c.f. [8]).	22	34.9	17	27.0	39	31.0
Incorrect Identifier Style	Not as `variableName`, `CONSTANT_NAME`, `ClassName`.	15	23.8	20	31.7	35	27.8
Incorrect Calculation	Implemented calculation yields an incorrect result.	21	33.3	13	20.6	34	27.0
Poor Identifier Names	Uninformative or confusing names are used for identifiers.	5	7.9	13	20.6	18	14.3
Incorrect Values	Incorrect values used as literals or in definitions.	4	6.3	7	11.1	11	8.7
Incorrect Classname	Class definition and/or filename does not match specification.	3	4.8	4	6.3	7	5.6
Exceeds Range	Index can exceed range of an array, list, or file.	3	4.8	4	6.3	7	5.6
Incorrect Input Validation	Enforces validation that rejects valid user inputs.	3	4.8	N/A	N/A	3	4.8
Misspellings in Strings	String literals and definitions contain misspellings.	6	9.5	0	0.0	6	4.8
Lack of Comments	Informative comments not included to explain some procedures.	3	4.8	2	3.2	5	4.0
Incomplete Implementation	Some requirements of the task are not implemented.	2	3.2	3	4.8	5	4.0
Incorrect Filename	Attempts to read file with incorrect name.	1	1.6	2	3.2	3	2.4
Missing Syntax Elements	Syntax elements (braces, string concatenations, etc.) are missing.	2	3.2	0	0.0	2	1.6
Logic Flow Error	Statements used in incorrect parts of an if-else statement.	2	3.2	0	0.0	2	1.6

Task 1 (T1) required the students to perform calculations on user input, process the contents of a text file, and print the results to the terminal in a column-based format. This task assesses a student's ability to implement simple input and output, along with developing a simple algorithm to perform a calculation. Task 2 (T2) had students render a 2D image consisting of various elements. Some elements were specified to be statically defined. Another was to be read from a file containing unicode characters representing pixel values, and rendered alongside a mirrored copy. Additionally, pixels were to be randomly set in part of the image. The task evaluates a student's usage of datatypes, loops, arrays and library calls. Students were required to submit executable main classes named `Assignment1.java` and `Assignment2.java` respectively, in order to simplify the marking process.

We also had access to the test data and model solutions written by the course's leader. For T1 we augmented this with manually defined inputs for specific edge cases, alongside randomly generated inputs within the task's domain.

B. Methodology

In order to identify mistakes that cause failures for both tasks, our script compiled and executed each student and model solution with the appropriate set of test data. Some solutions would not compile, or would encounter a runtime exception. In these cases, we noted the cause of the error, added a repaired variant to the solution set, and restarted the script. Our script stored the output of every execution, and compared it to the output of the corresponding model solution. We recorded notable differences between these outputs which suggested that a solution was not correct. In these cases, the solution contains a mistake that causes a failure.

We performed a manual analysis on each solution's source to locate mistakes which either cause the observed issues, or directly violate Java programming guidelines. When a new type of mistake was located, we checked previously analyzed solutions for it, in order to ensure that no instance of a given mistake class was missed. For each instance of a mistake, we recorded observations of the nature of its manifestation. These observations were used to inform our construction of a mutation operator to simulate the mistake.

C. Discussion

Table I shows the mistake classes that we observed in our dataset. The clearest observation that can be made from our classification is that there is a broad frequency range for the identified mistake classes. Our results show that there are considerably more code quality and style issues in submitted student solutions than faults that directly impact functionality. This supports the findings of Keuning et al. that quality issues tend to go unrepaired [9], indicating that these style and quality mistakes should be considered in autograding.

Our analysis differs from previous work on identifying student code mistakes via static analysis and compiler results [3], [9], since we consider each student's program individually, with knowledge of a correct solution's behavior and qualities. The dataset used in the previous work contains incomplete programs with mistakes that students may eventually fix. Conversely, our dataset consists of only final student submissions, revealing only mistakes that students have missed.

Our data shows that each mistake class tends to have a significantly different frequency for each task. It is likely that these frequencies are affected by both the nature of the task itself and the experience of the students, which they gain through feedback. The frequency of different mistakes in a task's solutions may inform a tutor when giving general feedback to students, and when developing other programming tasks to evaluate students for improvement in these areas.

Some mistake classes may require additional consideration when using an autograder. Solutions we observed that had "Missing Syntax Elements" (Table I) were not compilable, which should be reported during the grading process. However,

TABLE II
PROPOSED MUTATION OPERATORS FOR EACH MISTAKE CLASS

Citations indicate existing similar or equivalent mutation operators already proposed in the literature.

Mistake Class	**Mutation Operator**	**Example Mutant**	
		Pre-Mutation (Correct)	*Post-Mutation (Simulated Fault)*
Functional			
Incorrect Calculation	Replace arithmetic operators, or add new operators and values [5].	`n = 1 + 2;`	`n = 1 - 2;`
Incorrect Values	Replace a literal value with another value of the same type [5].	`double pi = 3.1416;`	`double pi = 4.7412;`

Exceeds Range	Modify the index when reading an array/list [5], or change the limit of a for loop that references an array.	`int[] array = new int[11];`
		`int n = array[10];`
			`int[] array = new int[11];`
			`int n = array[11];`

Incorrect Input Validation	Add an if statement that calls `System.exit()` if a random variable satisfies a random condition.	`int x = 100;`	`int x = 100;`
			`if (x > 8) {`
			`  System.exit(0);`
			`}`

Misspellings in Strings	Add, remove, replace, or transpose characters in strings (either literals or variables).	`String s = "hello␣world";`	`String s = "hello␣wolrd";`

Incomplete Implementation	Remove (or comment out) an output statement [5].	`System.out.print("Correct");`
		`System.out.println("Output");`
			`//System.out.print("Correct");`
			`System.out.println("Output");`

Incorrect Filename	Add, remove, replace, or transpose characters in the argument of a file reading method call.	`FileReader f = new FileReader(`
		`    "filename.txt");`
			`FileReader f = new FileReader(`
			`    "File.txt");`

Incorrect Classname	Add, remove, replace, or transpose characters in a class's declaration and/or filename.	`public class ClassName { ... }`	`public class classname { ... }`

Missing Syntax Elements	Remove a syntax element (string concatenations, braces, operators, comparators, parentheses, etc.)	`s = "concatenated" + n +`
		`            "string";`
			`s = "concatenated" + n`
			`            "string";`

Logic Flow Error	Move lines out of or into if/else blocks.	`if (cond) {`
		`  n = 2;`
		`}`
			`if (cond) {`
			`}`
			`n = 2;`

Quality			

Literal Value Repetition	Replace constant references with its value as a literal.	`final int WIDTH = 10;`
		`padString(forename, WIDTH);`
		`padString(surname, WIDTH);`
			`padString(forename, 10);`
			`padString(surname, 10);`

Statement Repetition	Replace a method call with its contents, expand a loop, or move a statement to each branch of a conditional.	`for (int i = 0; i < 3; i++) {`
		`  methodCall();`
		`}`
			`methodCall();`
			`methodCall();`
			`methodCall();`

Constants Defined as Variables	Remove the `final` keyword from a constant definition.	`final int CONSTANT = 8;`	`int CONSTANT = 8;`
Poor Identifier Names	Replace an identifier's name with an uninformative word.	`int length = 4;`	`int value = 4;`

Lack of Comments	Remove a comment line, multi-line comment, javadoc comment, or a comment at the end of a line.	`// Description of procedure`
		`...`
			`...`

Style			

Poor Indentation	Add or remove a random number of indents before a line. Replace some tabs with spaces, or vice-versa.	`if (cond) {`
		`  methodCall();`
		`}`
			`if (cond) {`
			`methodCall();`
			`}`

Overly Long Lines	Remove new-lines from a multi-line statement, or move a line to the end of the line preceding it.	`s = "long" +`
		`      ... +`
		`      "string";`
			`s = "long" + ... + "string";`

Incorrect Identifier Style	Change an identifier's capitalization, and/or remove or add underscores between words.	`int columnHeight = 24;`	`int Column_Height = 24;`

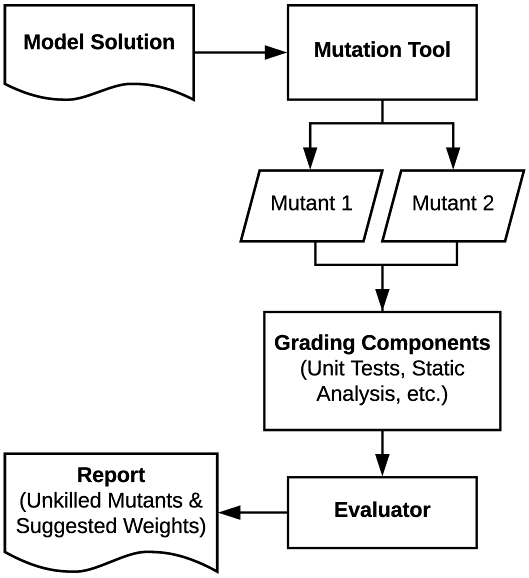

Fig. 1. Proposed evaluation approach overview

tests used to grade other functionality would also fail, such as those covering "Incorrect Calculation", resulting in an unfair grade if the missing syntax was the only actual mistake.

We observed that some mistakes contributed to the prevalence of others. For example, some cases where an array's index "Exceeds its Range" were caused by an "Incorrect Calculation". Another solution had "Poor Indentation" in a nested if-else, likely causing the student to make a "Logic Flow Error". This case illustrates the importance of encouraging adherence to style guidelines.

III. Mutation Operators

Table II shows the mutation operators that we defined using our observations and mistake classes. Each operator is listed with a description of the mutation process, and an example of the operator's effect on correct code. Some of these operators, such as introducing "Incorrect Calculation" (Table II), can be implemented using existing mutation operators [5]. This does not apply to the majority of our operators. Some of our operators uniquely impact functionality, such as that for "Missing Syntax Elements", which will cause a compiler error. This is not present in conventional operators, since detecting compiler errors is not necessary when evaluating a test suite, yet it should be considered when evaluating the fairness of autograding. Our other operators introduce poor style and code quality, which are not present in existing mutation operators that only focus on functionality.

IV. Proposed Technique

Figure 1 shows an overview of our proposed technique to apply these mutation operators in the evaluation of autograding configurations. A "Mutation Tool" (Figure 1) would apply our mutation operators to a task's tutor defined model solution, creating a set of "Mutants" for each simulated mistake class. The task's grader is executed on each of these mutants. This grader can be modeled as a set of individual "Grading Components", comprising of tests and static analysis processes. Every component that reports an error (e.g., a failing test) for a mutant is marked as "killing" the mutant [10].

An "Evaluator" (Figure 1) will provide the tutor with a "Report" based on the results of executing grading components with mutants. First, the evaluator lists mutants that are not killed by any components. This provides the tutor with knowledge of mistake classes that are potentially not covered by the grading configuration, allowing for such an issue to be resolved. Mutants of mistake classes that are not required for the task's learning objectives can be safely ignored.

Our evaluator would also provide a "Suggested Weight" for each grading component, so that each component appropriately influences the grade that a student receives. Conventional mutation analysis uses a mutation score of the percentage of mutants that are killed [7]. However, if several components each cover multiple mistake classes, an individual mistake class cannot be assessed. This leads to unfairness if every component is weighted equally or by a simple mutation score, since a student that makes one of these mistakes would be punished by all of these components failing. Instead, we will define a metric that favors grading components which only detect mutants of the same mistake class, rather than broad components that detect many. The weight of a component would be determined by the classes of mutants that it kills, the number of mutants in these classes that are killed, and how many other components kill the same mutants. This approach improves the fairness of the grading configuration, since it accounts for the differences between grading components.

The educator should also be able to flag some mistake classes as out of scope for a task, indicating knowledge that students have not yet been taught in the course. These flagged mutants can be considered in the weighting metric to reduce the impact on a student's grade when making a mistake of the same class.

V. Conclusions and Future Work

In this paper we have identified various student mistake classes through the observation of real student assignment solutions. We defined mutation operators that simulate each of these mistake classes. We also proposed a technique that applies mutation to improve the fairness, accuracy and completeness of autograding in beginner programming courses.

Our future work will be focused on developing this technique, including the implementation of our mutation operators and the evaluator itself. We plan to evaluate our technique by generating weights for each task in our dataset, and comparing the grades produced with and without these weights against the manually derived real grades of the dataset. We will also add operators for other mistake classes that are identified in existing work [3], [9], alongside any classes that we identify in future studies.

Acknowledgment

Phil McMinn is supported in part by the Institute of Coding, funded by the Office for Students (OfS), England.

References

[1] National Academies of Sciences, Engineering, and Medicine, *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. The National Academies Press, 2017.

[2] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, 2005.

[3] N. C. C. Brown and A. Altadmri, "Novice Java programming mistakes: Large-scale data vs. educator beliefs," *Trans. Comput. Educ.*, 2017.

[4] J. Breitner, M. Hecker, and G. Snelting, "Der grader Praktomat," *Automatisierte Bewertung in der Programmierausbildung*, 2017.

[5] R. A. DeMillo, D. S. Guindi, W. McCracken, A. J. Offutt, and K. King, "An extended overview of the Mothra software testing environment," in *Workshop on Software Testing, Verification, and Analysis*, IEEE, 1988.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *ICSE '05*, pp. 402–411, ACM, 2005.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *FSE 2014*, pp. 654–665, ACM, 2014.

[8] Google, "Google Java style guide." https://google.github.io/styleguide/javaguide.html#s4.4-column-limit. [Online; accessed 27-Sept-2018].

[9] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *ITiCSE '17*, pp. 110–115, ACM, 2017.

[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, Sept 2011.