A Tool for Mutation Analysis in Racket

Bambi Zhuang
Northeastern University
Boston, USA
zhuang.ba@northeastern.edu

James Perretta
Northeastern University
Boston, USA
perretta.j@northeastern.edu

Arjun Guha
Northeastern University
Boston, USA
a.guha@northeastern.edu

Jonathan Bell
Northeastern University
Boston, USA
j.bell@northeastern.edu

***Abstract*—Racket is a functional programming language that is used to teach CS1 at many high schools and colleges. Recent research results have shown that mutation analysis can be an effective substitute for manual grading of student test cases. In order to evaluate its efficacy in our college's introductory programming courses, we created a prototype mutation analysis tool for Racket. We describe the design and features of the tool and perform a feasibility study using two assignments from an intro CS course where student test suite thoroughness was evaluated by hand by human graders. In our results, we find a moderate correlation between mutation score and hand-grading test suite quality score and conduct a qualitative analysis to identify situations where mutation score and hand-grading score do not correlate. We find that, compared to hand-grading, mutation analysis may require more stringent adherence to the interface specified in an assignment as well as more precisely specified assignments. On the other hand, inter-reviewer reliability is a known challenge of hand-grading, and we observe several instances where hand-graders may have assigned the wrong score. Given the relatively cheap cost to providing mutation analysis feedback to students (compared to hand-grading feedback), mutation analysis still provides the opportunity to provide faster, more frequent, feedback to learners, enabling them to improve their testing practices further. Future work will study the effectiveness of various mutation operators in Racket and perform larger-scale evaluations.**

***Index Terms*—Racket, mutation testing, mutation analysis, mutation analysis tool**

I. Introduction

Racket is a mostly-functional programming language that is a modern dialect of Lisp and a descendant of Scheme. Its simplicity, teaching-language packs, and expression-focused syntax contribute to Racket's use in computer science education at several prestigious undergraduate programs, including our own. The curriculum, based on the text *How to Design Programs* integrates education on software testing [1]. *DrRacket*, the IDE that accompanies the language, shows line coverage using color-coded highlights and supports many different methods of unit testing. Automated grading scripts check the functional correctness of student code, and *DrRacket* provides code coverage metrics of student test cases on their own implementation, but code coverage has known limitations. To date, there is no more-robust, automated approach for grading the quality of student test cases in Racket.

This work is partially supported by a Khoury College Teaching Innovation Grant and National Science Foundation grants CCF-2102288, CCF-2100037 and CNS-2100015.

Recent research results have shown that mutation analysis is an effective substitute for manual grading of student test cases [2]. In this approach, programming assignments require students to implement a standardized interface along with tests for their interface. In order to grade the students' test cases, the grading server performs mutation analysis on an instructor-written solution, identifying how many mutants are detected by each student's test suite.

Popular, existing mutation analysis tools include PIT and Jumble for Java; Stryker for JavaScript, C#, and Scala; Mull for C++. These tools share the well-researched premise of how the most effective test suites for detecting trivial faults, such as flipped numerical comparisons, also best detect higher-level faults. MuCheck implements common mutation operators for Haskell [3], [4]. However, we are unaware of any other off-the-shelf mutation analysis tools for other functional languages or for Racket. Functional languages present many opportunities for new mutation operators, and determining the ideal set of mutation operators to use is an interesting research challenge.

To fill this gap, we present MACKET , a mutation analysis tool for Racket. We perform a preliminary evaluation of MACKET using a set of programming assignments and student test cases from a course at our institution. During that semester, each student test suite was manually graded by a teaching assistant. We compare the mutation scores of each test suite with their manual grading scores in an effort to determine whether our mutation operators and the tool could be used as a substitute for hand grading in future semesters. We seek to answer the following research question:

1) **RQ**: Is mutation analysis a good substitute for manual grading of test suite quality in Racket?

II. A tool for mutation analysis in Racket

MACKET is a mutation analysis tool designed to support mutation analysis for Racket programs and more generally the use of mutation analysis in pedagogical contexts. We describe a few key aspects of its design, including a list of mutation operators we implemented.

A. Design and Tool Usage

MACKET uses a multi-phase process in which mutants can be generated before they are executed. Using this model, instructors can generate the mutants, determine which mutants should be used for grading, and then specify which mutants to include or exclude during the mutant execution phase. Prior to

the mutant execution phase, MACKET supports checking for and filtering out test cases that contain false positives (i.e., tests that fail when run against a correct implementation). This allows students to receive partial credit and feedback on the rest of their test cases, and distinguishes it from other mutation tools that require all tests to pass on the (not mutated) system under test. The interface for the mutant execution phase is language-agnostic, allowing for future extension to other languages. In theory, a user could generate mutants with another tool and run them using MACKET . MACKET produces JSON- and HTML-formatted results using the same schema and display frontend used by Stryker Mutator [5].

B. Mutation Operators

In selecting which mutation operators to implement, we draw from the mutation analysis literature and state-of-the-practice mutation analysis tools for imperative languages such as PIT [6] and Stryker [7]. Since functional languages such as Racket do not have certain control-flow structures (such as loops) common to imperative languages (relying instead on programming techniques such as recursion and higher-order functions), our first step was to determine which imperative mutation operators have a meaningful equivalent in Racket. For example, some mutation operators, such as replacing arithmetic operators, have a clear mapping from an imperative language to Racket. Others operators, such as statement deletion, do not have an immediately obvious mapping because Racket is an expression-based language that does not have statements. We selected a subset of standard mutation operators for imperative languages that have a clear mapping from their imperative to functional versions. We also implemented two operators specific to Racket and/or functional languages in general.

Operators Based on Standard Imperative Operators:

- ArithmeticMutator: switching between +, −, ⋆, / operators
- LogicalMutator: switching between `AND` and `OR`
- IfMutator: changing the predicate of `if` to `True` and `False`
- CondMutator: changing the predicate(s) of the conditional to `True` and `False`
- WrapWithNotMutator: inverting any boolean return by wrapping with the `not` operator
- FlipNumSignMutator: flipping the sign of numeric literals
- ArithmeticDeletionMutator: deleting parts of arithmetic operations and replacing with the arithmetic parts (ex. `(+ a b)` → `a`)
- NumberComparisonMutator: switching between <, <=, ==, >=, > comparisons
- FlipBooleansMutator: changing `True` to `False` and vice versa
- NumLiteralsMutator: replacing numeric literals with `0`, `-1`, `1`, `value+1`, `value-1`
- EmptyStringMutator: replacing strings with empty strings
- BoolParamsToBool: replacing the parameters of `AND`, `OR`, `NOT` with `True` and `False`

Functional- or Racket-specific Operators:

- BoolFuncToBoolMutator: A mutator that contains a list of known boolean-return methods in the language such as `andmap`, `ormap`, and comparison methods and replaces them with `True` or `False`. It also finds calls to methods whose names end in `?`, which is a Racket convention for bool-returning methods, and replaces those calls with `True` or `False`.
- EmptyListMutator: Replacing lists with empty lists. We note that Le et al. [3] implemented this operator for Haskell.

Non-applicable Imperative Operators Due to the available language features of Racket, the following is a list of non-applicable mutator operators that we did not implement:

- Stryker: Block Statement, Optional chaining, Regex [7]
- PIT: Void Method Calls, Null returns, Constructor Calls, Remove Increments [6]

a) Other Unused Operators: We wrote one homework-specific mutator that swapped calls to similar methods (e.g., `fold-left` & `fold-right`); however, that assignment did not end up being graded for test suite quality. Future work might consider evaluating such mutators in the context of student test cases.

III. EVALUATION METHODOLOGY

Our goal in this study is to examine the feasibility of evaluating test suite quality in intro CS assignments written in Racket using mutation analysis compared to human hand-grading. We collected mutation scores and hand-graded test suite quality scores from two programming assignments and examined the correlation between these scores. We also manually inspected the test suites and conducted a qualitative analysis on any outliers. We describe the datasets we collected in more detail below.

A. Datasets

We collected data from two assignments from one semester of an intro CS course. In order to address our research questions, we required the following information:

1) Student test suites, which were graded by hand and assigned a discrete score reflecting the thoroughness of the test suite. The graders were graduate-level teaching assistants who were responsible for writing the hand-grading rubrics with the oversight of the course instructor.
2) Mutation scores for those test suites, computed using MACKET
3) Mutation scores for the instructor-written test suites, which were also used to evaluate student implementation correctness.

We selected programming assignments where student test suite quality was graded by hand and where the abstractions being tested were well-specified. That is, student test suites should behave correctly when run against other implementations. We examined a total of 56 assignment submissions

across two programming assignments. We collected only one submission per student for each assignment. Here we briefly summarize each assignment and their grading rubrics:

List abstractions. Students were required to implement 5 list-processing functions, e.g., interleave, intersection, earliest, and write test cases for those functions. We collected 29 submissions. Test cases were graded on a discrete scale of (0, 1, 1.5, 2), with 0 meaning "No tests present," 1 meaning "Incomplete coverage," 1.5 meaning "Missing edge cases," and 2 meaning "Tests correct."

Self-Referential Data. Students were required to implement two basic binary-tree traversal functions and write test cases for those functions. We collected 27 submissions. Test cases were graded on a discrete scale of (0, 1, 2), with 0 meaning "No tests present," 1 meaning "Incomplete coverage," and 2 meaning "Tests correct."

We ran MACKET on all the student test suites we collected, generating mutants from the instructor-written solution, and collected mutation scores for each test suite. When collecting mutation scores for student test suites, we first discarded individual test cases that displayed false positives when run against a correct instructor implementation. We note that hand-graders may not have performed such a step, as they did not run the student test suites. We discarded mutants applied to parts of the instructor solution that students were not required to test. We also discarded mutants that were exact source code duplicates of each other. We then looked for a correlation between hand-graded test quality scores and mutation scores. We also conducted a qualitative analysis of submissions with high hand-graded scores but low mutation scores and vice-versa, looking for instances where the hand-grading rubric was applied incorrectly or where additional mutation operators may be required in order to compute a more accurate mutation score.

IV. RESULTS

We conduct an analysis of the data we collected from these two programming assignments.

A. Assignment 2: List Abstractions

In Figure 1, we see a moderate correlation between mutation score and hand-graded test suite quality score (Pearson r=0.51). MACKET generated a total of 164 mutants, and we discarded 28 that were exact duplicates and 82 that were applied to parts of the instructor implementation that students were not required to test. The instructor-written test suite achieved a mutation score of 79.63%, but we were able to increase that score to 94.44% by adding additional test cases. The lower instructor test suite mutation score appears to be caused by there being no test cases for one of the methods students were required to implement.

We observe several instances where a student test suite's hand-graded test suite quality score does not align with that test suite's mutation score. For example, we see five (lower dots at hand grade 10) student test suites that achieved the maximum possible hand-grading score despite having mutation scores as low as 81.48%. We also see a (bottom dot at 7) student test suite with a hand-grading score of 7/10 despite having a low mutation score of 55.55%. Furthermore, we see two (dots left of 7) student test suites with hand-grading scores less than 7 despite having mutation scores of at least 75%. We note further that among student test suites with a hand-grading score of 7, we see a wide range of mutation scores, as low as 55.55% and as high as 92.59%.

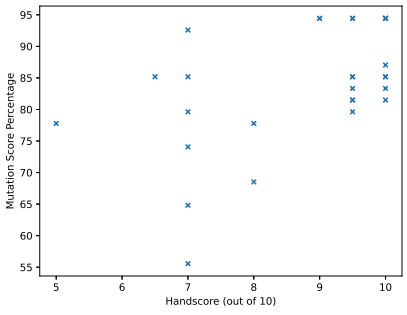

Fig. 1. Plot of mutation score versus hand score for Assignment 2. We see a moderate correlation (Pearson r=0.51). The maximum hand-grading score possible was 10 points (two for each of the five methods) and the maximum possible mutation score was 94.4 due to equivalent mutants. After manual analysis, most of the visual outliers can be explained by either a decreased mutation score due to discarded tests (from under-specification of the assignment) or to an inaccurate hand score (from inconsistencies between hand graders).

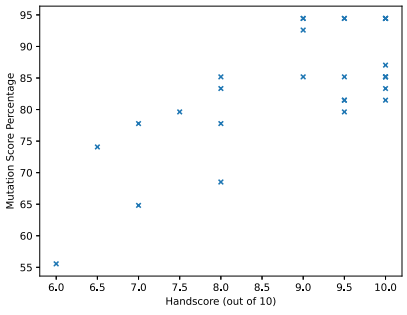

Fig. 2. Plot of mutation score versus hand score for Assignment 2 after updating hand scores to correct for inconsistent applications of the hand-grading rubric. We see a strong correlation (Pearson r=0.73). The maximum hand-grading score possible was 10 points (two for each of the five methods) and the maximum possible mutation score was 94.4 due to equivalent mutants (out of 54 mutants total after filtering).

We conducted a manual qualitative analysis of every student test suite and adjusted hand-grading scores to correct inconsistent applications of the rubric. Figure 2 contains the updated results. After adjusting the hand-grading scores, we see the correlation between mutation score and hand-graded test suite quality score increase to r=0.73.

a) Qualitative Analysis: We manually inspected every student test suite for this assignment to identify why some hand-grading and mutation scores did not correlate. We found

seven test suites that contained at least one test case that we discarded due to the presence of false positives when run against a correct instructor implementation. However, all of these false positives were due to under-specification of the function being tested, and hence did *not* receive any penalty for this in the manual grading. For example, 7 of these test suites expected the `powerlist` function to return elements in reverse order from what the instructor suite implemented (the order of returned list elements was not specified in the assignment description). The eighth test suite expected that `earliest` function's arguments should be placed in a different order (while the assignment description implies the order of arguments, it does not precisely specify it). In all of these cases, the hand-graders appear to have allowed for these variations in cases where the behavior is under-specified.

We also identified two test suites where hand-graders may have assigned too high a score and five where hand-graders may have assigned too low a score. We determined this by manually examining the test suite code and identifying situations where similar test suites received different scores. In each of these cases, we determined what the correct score should be and updated those scores in our dataset before recomputing the correlation.

b) Assignment 3: Binary Tree Traversal: The correlation between mutation score and hand-graded test suite quality score for this assignment is undefined. In Figure 3 we see that all student test suites received the same hand-graded test suite quality score. We see a mutation score of 100% for all student test suites except for one, which has a mutation score of 0%. In the one instance where a student test suite did not detect any mutants, this was because the student's tests passed arguments to the method being tested in reverse order, which caused those tests to display false positives and be discarded by our tool. The hand-graders appear to have allowed for this variation, despite the order of parameters being well-specified in the assignment instructions. MACKET generated a total of 136 mutants, and we discarded 8 that were exact duplicates and 121 that were applied to parts of the instructor implementation that students were not required to test. Interestingly, we found that the instructor test suite failed to detect one mutant that all but one of the student test suites detected. This mutant replaced a call to `plus` with one of its operands in the base case of one of the tree traversal methods.

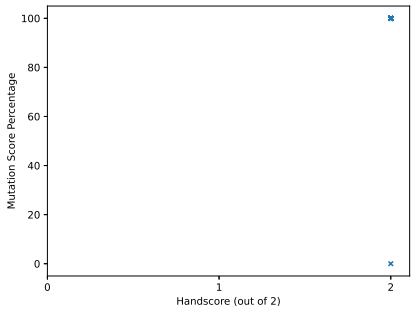

Fig. 3. Plot of mutation score versus hand score for Assignment 3. We see an undefined correlation between hand-graded test suite quality score and mutation score because all the student test suites received the same hand-grading score. The two methods were graded together with a maximum hand score of 2, and the maximum possible mutation score was 100% (out of 7 mutants total after filtering). The single point with a mutation score of 0% and a hand score of 2 is explained by the student violating assignment specifications, resulting in all their test being thrown out due to false positives by our tool.

V. DISCUSSION

We discuss the implications of our results for software testing researchers, software testing educators, and mutation analysis tool builders. We also reflect on the threats to the validity of our conclusions and the efforts that we took to mitigate those threats.

A. Implications for Researchers

The results of our study suggest that evaluating test suite quality using mutation analysis is feasible in intro CS courses using Racket. Future work can examine the efficacy of mutation analysis on a wider range of programming assignments and explore potential new mutation operators for Racket. We note the importance of making sure assignments are well-specified when conducting mutation analysis on student test suites. Many of the mutation score outliers we observed were caused by students testing an implementation that was allowed by the assignment specification but different from the instructor's implementation.

B. Implications for Educators

Our study reveals certain trade-offs between mutation analysis and hand-grading of student test suite quality. While human hand-graders are able to be more flexible in their grading, they may be less consistent in their application of the grading rubric. We observed several instances where hand graders appear to have applied the grading rubric inconsistently. For example, deductions were applied to most students who didn't have sufficient tests for the method `earliest`; however, one student who didn't have any explicit tests received almost full credit. We note that the correlation between mutation score and hand-graded code quality score increased significantly after correcting for inconsistent application of the hand-grading rubric. It may be possible to improve human grader accuracy by providing them with the mutation analysis results, and future work may explore this question. Mutation analysis produces more consistent results in this regard but requires that the interfaces being tested be well-specified. On the other hand, mutation analysis can also check test suites for correctness as well as quality, which may be harder for human graders to do by hand. We note that precise assignment specifications are likely more desirable as class size increases. Mutation analysis can also potentially be used to provide frequent, actionable feedback to students on the quality of their test suites.

We also observe that mutation analysis can be useful to instructors when examining the quality of their own test suite.

Knowing which mutants are undetected, instructors can make an informed decision on whether to strengthen their test suite or exclude certain behaviors from grading. For example, we discussed the single mutant undetected by the instructor test suite for Assignment 3 with the instructor, who agreed that the mutant is a case that students should be expected to write tests to detect, although it sparked an interesting discussion of whether it would be coupled with other mutants. Future work should investigate coupling effects in the context of testing education.

C. Implications for Tool Builders

We believe that supporting phase separation and including a more fine-grained false positive check are useful features for pedagogical applications of mutation analysis. The default behavior of traditional mutation analysis tools (e.g. PIT [8] or Stryker [5]) is to require all tests to pass on the system under test before mutation is performed. However, by filtering out tests with false positives rather than aborting if any false positives are present, students can be given feedback on the thoroughness of their remaining correct tests as well as feedback on which tests contain false positives. Additionally, being able to generate mutants ahead of time makes it possible to filter out equivalent and trivially detectable mutants and run student tests against only the remaining mutants. Other advanced mutation analysis tools, such as MuCPP [9] also pre-generate mutants, but this technique is largely unsupported by state-of-the-practice Pitest [8], Stryker [5] and mull [10]. Future work might specifically examine different use cases for pre-generating and filtering mutants before attempting to execute them.

D. Threats to validity

a) Construct: Are we asking the right questions?: Our research questions are based on established research questions from the mutation analysis literature. We posed our research question before we examined our dataset. These questions were prompted by our experience evaluating student test suite quality by hand and evidence that students benefit from receiving frequent, actionable feedback.

b) Internal: Do our methods and datasets affect the accuracy of our results?: Our research question requires assignments where student test suites were graded by hand for thoroughness and where the assignment specification was precise enough to support mutation analysis. Because of this, we were only able to include two assignments, and both of these assignments had portions where we could not conduct mutation analysis.

There could be bugs in MACKET and the other scripts we wrote. We carefully examined the output of each step in our analysis and investigated discrepancies. By manually inspecting the source code of all of the student test suites, we were able to distinguish between bugs in our tools, hand-grader errors, and assignments being under-specified. We corrected all of the bugs we found in our tools before finalizing the data-sets.

c) External: Would our results generalize?: Our evaluation uses two programming assignments, and they may not be representative of programming assignments in Racket used in other intro CS courses. We are careful to avoid claiming that our results will generalize, and share them as a work-in-progress. Future work will conduct a similar analysis on additional Racket programming assignments.

VI. Related Work

Most existing mutation analysis tools support imperative languages such as Java [8], JavaScript/TypeScript [5], C# [5], and C++ [10]. While Lazarek's "Mutate" library provides an API for defining mutation operators in Racket, we are not aware of any "out-of-the-box" mutation analysis tools for Racket other than MACKET . Le et al. [3] have studied mutation analysis in functional programming languages such as Haskell, and there are also several mutation analysis tools for CoQ [11], [12].

Le et al. describe three functional mutation operators in particular: reordering pattern matching, which the assignments we analyzed did not use; mutation of lists, of which we implemented the "list identity" portion; and type-aware function replacement, which we only implemented heuristically for bool-returning functions, as Racket is dynamically typed.

There is a growing body of work on the effectiveness of mutation analysis for evaluating student test suite quality. Jia and Harman present a survey on mutation analysis [13], which Just et al. [14] show is correlated with real-fault detection, even after controlling for coverage. Other work examines the extent to which mutants are coupled to real faults in student-written code or manually-seeded faults written by an instructor [2], [15]. Code Defenders [16] is an interesting example of how teaching software testing can be enhanced with gamification, and perhaps there is future work that could explore the use of mutation analysis tools in such a context. The effectiveness of mutation analysis depends on the kinds of mutants generated, and there are several ways to improve the mutant generation process [17], [18]. Our study suggests that many traditional mutation operators for imperative languages can be effectively adapted and used in functional languages and also implements one mutation operator specific to functional languages and one operator specific to Racket.

VII. Conclusion

Mutation analysis can be an effective approach for providing automated feedback to students about the quality of their of their test cases. We implemented MACKET , a tool for mutation analysis of code written in Racket, a functional language that is used in introductory programming courses at several universities including our own. Our pilot study examined the question of whether MACKET is a good substitute for manual grading of test suites written in Racket. These preliminary results were quite positive, demonstrating a moderate correlation between mutation score and hand-grading score on one of the two assignments. Our qualitative analysis of the

submissions revealed several interesting implications for educators interested in providing automated feedback on students' test case quality. Our ongoing work with MACKET includes evaluating additional mutation operators and deploying the tool to larger classes to gain more feedback. We have released MACKET under an open-source license, and we look forward to collaborating with colleagues at other institutions on its evaluation [19]. Future work may examine related questions such as examining the productivity of the mutation operators we implemented and designing mutation operators based on real student faults found in student submissions.

ACKNOWLEDGEMENTS

We thank the teaching assistants and lecturers for sharing course data with us and for providing insights and guidance while we were processing the assignment data. We thank Cameron Moy for consulting about the Racket language while we implemented our tool.

REFERENCES

[1] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018.

[2] J. Perretta, A. DeOrio, A. Guha, and J. Bell, "On the use of mutation analysis for evaluating student test suite quality," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 263–275. [Online]. Available: https://doi.org/10.1145/3533767.3534217

[3] D. Le, M. Amin Alipour, R. Gopinath, and A. Groce, "Mutation testing of functional programming languages," Oregon State University, Tech. Rep., 2014.

[4] D. Le, M. A. Alipour, R. Gopinath, and A. Groce, "Mucheck: An extensible tool for mutation testing of haskell programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 429–432. [Online]. Available: https://doi.org/10.1145/2610384.2628052

[5] "Stryker mutator," 2022. [Online]. Available: https://stryker-mutator.io/

[6] "Pit overview," 2022. [Online]. Available: https://pitest.org/quickstart/mutators/

[7] "Stryker supported mutators," 2022. [Online]. Available: https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/

[8] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

[9] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for c++ with the mucpp mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916301161

[10] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on llvm," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2018, pp. 25–31.

[11] "Quickchick," 2023. [Online]. Available: https://github.com/QuickChick/QuickChick

[12] M. Cavada, A. Col'o, and A. Momigliano, "Mutantchick: Type-preserving mutation analysis for coq," 2020. [Online]. Available: https://ceur-ws.org/Vol-2710/short2.pdf

[13] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, 2011.

[14] 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 *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 654–665. [Online]. Available: https://doi.org/10.1145/2635868.2635929

[15] B. S. Clegg, P. McMinn, and G. Fraser, "An empirical study to determine if mutants can effectively simulate students' programming mistakes to increase tutors' confidence in autograding," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1055–1061. [Online]. Available: https://doi.org/10.1145/3408877.3432411

[16] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 162–167.

[17] P. Delgado-Pérez, L. M. Rose, and I. Medina-Bulo, "Coverage-based quality metric of mutation operators for test suite improvement," *Software Quality Journal*, vol. 27, no. 2, pp. 823–859, jun 2019.

[18] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 284–294. [Online]. Available: https://doi.org/10.1145/3092703.3092732

[19] B. Zhuang, J. Perretta, A. Guha, and J. Bell, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7689559