Who Tests the Testers?*

Avoiding the Perils of Automated Testing

John Wrenn
Computer Science
Brown University
USA
jswrenn@cs.brown.edu

Shriram Krishnamurthi
Computer Science
Brown University
USA
sk@cs.brown.edu

Kathi Fisler
Computer Science
Brown University
USA
kfisler@cs.brown.edu

ABSTRACT

Instructors routinely use automated assessment methods to evaluate the semantic qualities of student implementations and, sometimes, test suites. In this work, we distill a variety of automated assessment methods in the literature down to a pair of assessment models. We identify pathological assessment outcomes in each model that point to underlying methodological flaws. These theoretical flaws broadly threaten the validity of the techniques, and we actually observe them in multiple assignments of an introductory programming course. We propose adjustments that remedy these flaws and then demonstrate, on these same assignments, that our interventions improve the accuracy of assessment. We believe that with these adjustments, instructors can greatly improve the accuracy of automated assessment.

CCS CONCEPTS

• **Social and professional topics** → **Student assessment**; CS1;
• **Software and its engineering** → **Software defect analysis**;

ACM Reference Format:
John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?: Avoiding the Perils of Automated Testing. In *ICER '18: 2018 International Computing Education Research Conference, August 13–15, 2018, Espoo, Finland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3230977.3230999

1 INTRODUCTION

Instructors routinely rely on automated assessment methods to evaluate student work on programming assignments. In principle, automated techniques improve the scalability and reproducibility of assessment. However, while more reproducible than non-automated methods, automated techniques are not, ipso facto, more accurate. Automated techniques also make it easy to perform *flawed* assessments at scale, with little feedback to warn the instructor. Not only does this affect students, it can also affect the reliability of research that uses it (e.g., that correlates against assessment scores).

*This work was partially supported by the US National Science Foundation.

ICER '18, August 13–15, 2018, Espoo, Finland

ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3230999

In this work, we explore methods for assessing implementations and test suites submitted in response to programming problems. In particular, we consider how student-submitted artifacts may be used to enhance instructor-provided ones within the context of automated assessment. This is hardly a new question: as discussed in section 2, many authors use student artifacts to assess other students' work. However, we find that the models in the literature for doing this can have significant flaws that can unfairly reward or penalize students.

As we will show, the key to including student artifacts in a fair way builds on screening them with particular kinds of instructor-provided artifacts, both implementations and test suites, both correct and incorrect. Concretely, we analyze two common methods for assessing student implementations. We explore the methods both foundationally and experimentally, using data from an introductory course. We highlight the perils of these approaches, and present an improved model and technique with which instructors can immunize their assessments from these perils.

The contributions of this paper are:

(1) identification of conceptual pathologies in existing methods for automated assessment,
(2) experimental evidence that these issues arise in practice, and
(3) a new method for assessing implementations and test suites that mitigates these pathologies.

After reviewing related work (section 2) and defining terminology (section 3), we present (section 4) three models for assessing implementations (one of them novel). Section 5 describes a process for instructors by which our novel model can be combined with another in a manner that iteratively improves the outcomes of both until they are identical, and section 6 evaluates these models and this process experimentally in the context of assessing both implementations and test suites. Section 7 discusses implications for those who develop or use automated assessments for programming assignments.

2 RELATED WORK

Automatic assessment of student implementations and test suites is typically done by testing their behavior against a reference artifact (rather than through proof-based formal methods [31]). We focus our work (and thus this section) on assignments for which the inputs and outputs are data values (as opposed to, say, GUIs which require their own style of testing techniques [12, 17, 35]).

There are multiple choices for both the form of the reference artifact(s) and the corresponding testing methodology, depending on which student artifact one wishes to test.

2.1 Evaluating Implementation Correctness

Goldwasser [18] asked students to submit a collection of interesting inputs. He then ran each input through each of the student implementations and an instructor-written one, checking whether the two agreed on their computed output. He notes the challenge of this approach when the outputs are non-deterministic.

Many testing frameworks support assertions that consist of conditions to check against the run-time behavior of an implementation. While such assertions can be embedded in the implementation itself, we focus here on ones that are provided as a standalone artifact (as this is a better fit for automated testing). These assertions can check that a specific input yields a specific output, or that the output of a given function always satisfies a stated predicate (such as lying within a range of numbers). Assertions are part of most unit-testing frameworks; some languages even include constructs for these assertions directly in the language itself (e.g., Pyret [5], the Racket student languages [14], and Rust [6]).

Some forms of assertion-based testing generate the inputs to use in testing, rather than require students or instructors to provide them manually. Tools such as QuickCheck [3] generate test cases from formal specifications of a program's expected behavior (then test the program against the same formal specification).

Many instructors assess student implementations using a test suite of their own creation [2, 15, 16, 19, 21, 22, 24, 27, 36]). This approach is supported by major automatic assessment tools, such as ASSYST [23], Web-CAT [7], and Marmoset [34]. Some instructors also leverage student-written tests for testing other students' implementations. In the literature, this approach is most closely associated with all-pairs style evaluations, in which student test suites and implementations are assessed by running every test suite against every implementation [11, 18, 25]. This approach also appears in research on students' testing abilities, such as Edwards' proposed metric of "bug revealing capability" [8–10]. Broadly, student test suites can be appropriated for the task of assessing *any* corpus of implementations whose correctness is unknown—not just those of students. For instance, Shams and Edwards [30] use student test suites to filter out mutations of an initially-correct reference implementation whose faultiness is not detectable by any student or instructor test suite.

2.2 Evaluating Test Suites

Student test suites are typically assessed against two metrics: whether the tests conform to the specification (*correctness*), and whether the tests cover the interesting inputs to a problem (*thoroughness*) [27]. Assessing correctness of a test suite typically entails running it against an instructor-written implementation [2, 8–10, 27, 32]. This check is particularly important when using student tests to assess each others' implementations [8–10, 25].

Code coverage is often used as a proxy for thoroughness; ASSYST [23], Web-CAT [7], and Marmoset [34] all take this approach. Code coverage is attractive because it reflects professional software engineering practice [26] and is not labor-intensive [8]. However, a growing body of evidence challenges the appropriateness of coverage as a measure of thoroughness [1, 9, 20], in both professional and pedagogic contexts. Alternatively, instructors may run student test suites against a corpus of incorrect implementations, checking what fraction of these a test suite rejects. This corpus may be sourced from students [10, 11, 18, 25, 33], from machine-generated mutations of a reference implementation [1, 30, 33], or crafted by the instructor [2, 27].

3 ASSUMPTIONS AND TERMINOLOGY

We assume that instructors assess implementations by running tests against them, where each test indicates both an input to the program and the expected output (whether directly or via some sort of assertion). We do not assume that instructors are trying to handle all forms of assessment automatically; style and design assessments, for example, may be handled through separate processes and are out of the scope of this paper. This paper focuses on automated assessment for functional correctness. We further assume that instructors are willing to perform some manual inspection of some testing results as part of calibrating the artifacts against which automation will assess student work.

We will use the term *conforms* to describe test suites or implementations that are consistent with a given specification (usually provided by the problem statement). For a test suite to accurately flag non-conformant implementations, it needs to be fairly thorough (a term we introduced in section 2.2). Our definition of thoroughness suggests that it targets a relative, rather than absolute, standard. Completely thorough test suites are generally not achievable in practice: most programs have an infinite number of behaviors, which cannot be covered in a finite number of tests. Nevertheless, we assume that instructors are trying to be thorough relative to the bugs that are likely in student implementations. We call tests that are nonconformant (either because they assert something nonsensical, or they mis-represent the specification) *invalid*.

When assessing an implementation against a test suite, we say that the test suite *accepts* the implementation if every individual test in the suite passes on the implementation. If even one test fails, we will say that the test suite *rejects* the implementation.

Given a set of test suites to check an implementation, we will say (par abus de langage) that the implementation is *correct* (relative to those suites) if every test suite accepts the implementation. Otherwise, we will say that the implementation is *faulty*.

4 MODELS OF ASSESSMENT

In this paper we study and contrast three models for assessing student implementations, the first two of which are commonly used in prior work. We give each a name and describe its general form, though of course individual uses of each model may differ slightly. Figure 1 pictorially summarizes our models and the workflows that define them. The upper part shows a student implementation running against one or more test suites. The lower part shows which implementations a student test suite must pass to be run against other student implementations.

We study these models in two contexts: (1) assessing student implementations, and (2) assessing student test suites. Each model outputs a judgment of whether each student implementation is faulty or not. Having established the correctness of implementations, an instructor may then evaluate the accuracy of each student's test suite by checking how closely its judgments of implementation correctness match the judgments made by the model.

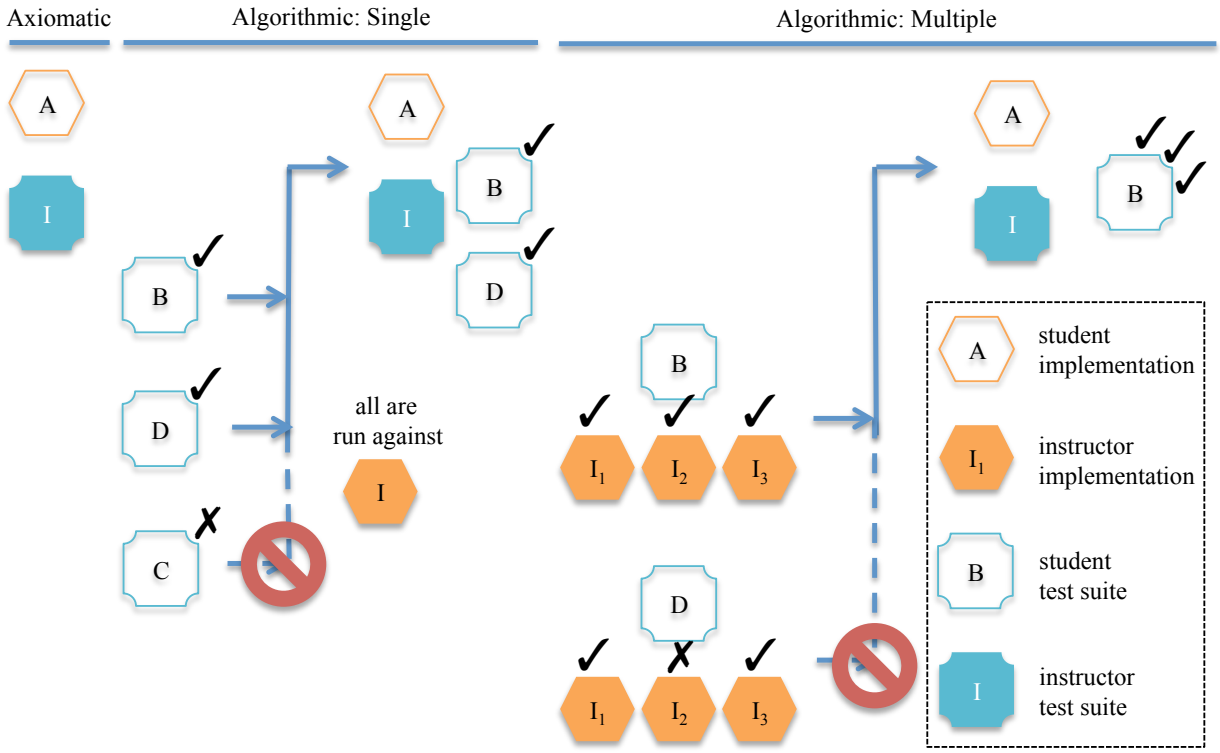

Figure 1: Three models of assessing implementations, each based on a different collection of test suites. Hexagons are implementations, squares with concave corners are test suites. Solid artifacts (labeled I) are instructor-provided while hollow ones are from students, with letters and numbers differentiating them as needed.

```
import number_sort from impl

check:
  number_sort([3,2,1,0]))
    is [0,1,2,3]
end
```

Instructor Test Suite

```
import number_sort from impl

check:
  number_sort([3,2,2,1]))
    is [1,2,2,3]
end
```

A Clever Test Suite

```
import number_sort from impl

check:
  number_sort([0,1,2,3]))
    is [3,2,1,0]
end
```

A Faulty Test Suite

Figure 2: Three contrived test suites for (ascending) `number_sort`: (1) an instructor test suite; (2) a student test suite that, by checking with an input that includes duplicate elements, can catch a bug that the instructor test suite cannot; and (3) a student test suite that, by expecting the sort to occur in descending order, is invalid with respect to the assignment specification.

4.1 Axiomatic (Axm)

The *axiomatic* approach, shown at the top left of fig. 1, is the simplest and most common:

> **Model Summary:** Each student implementation is assessed against a single instructor-written test suite.

In the figure, student A's implementation is being assessed against the instructor test suite.

Pitfall. This method relies solely on the judgment of the instructor test suite. If this test suite is incorrect (which it usually isn't, but could be in subtle ways), conformant student implementations may be labeled as faulty. What is even more likely, we contend, is that the instructor test suite may be insufficiently thorough, in which case student implementations that are actually nonconformant may be wrongly labeled correct.

As a simplistic but illustrative example, assume that students have been asked to implement a function, `number_sort`, that sorts its input in ascending order. Figure 2 shows a rudimentary instructor test suite (left). Another test suite (center) tests something beyond the instructor suite (in this case, correct handling of duplicate elements); we call such test suites *clever*. If a student implementation did not handle duplicate elements, the instructor test suite will accept it while the clever suite rejects it. The test suite on the right violates the specification by expecting descending order; such a test suite is invalid (as defined in section 3).

Instructors might assume (based on their experience with the material) that their test suites are correct and fairly thorough (especially for assignments they have given multiple times), but this model provides no inherent mechanism for validating this belief. The authors of this paper, in particular, were victims of this hubris (we return to this in section 6.1).

Assessment Impact. If the instructor test suite is insufficiently thorough, faulty implementations may be labeled as correct. These students would not receive feedback about these undetected flaws. Furthermore, if these labels are used as a basis to assess student test suites, students who manage to detect bugs the instructor does not will be penalized, since their suite's judgments about correctness disagree with the judgments of the instructor's suite.

4.2 Algorithmic: Single Implementation (AlgSing)

How might we raise the thoroughness of the test suite used for assessing student implementations? Numerous existing tools and methodologies augment instructor test suites with student-written ones (e.g., [8–10, 18, 30]). Of course, student test suites could be incorrect, so this approach needs a method to determine which ones can be trusted for this task. Validating students' test suites against an instructor implementation is a obvious (and oft-taken) approach:

> **Model Summary:** Each student implementation is assessed against the instructor test suite as well as student test-suites that are correct relative to the instructor implementation.

In the first test-suite assessment in fig. 1 (lower left), B's, C's, and D's test suites are evaluated against the instructor implementation. B's and D's are consistent with the implementation—the check marks on them indicate that they have passed this check—but C's is not. C's test suite is subsequently ignored, but B's and D's are added to the pool of test suites against which A's implementation is tested. The instructor test suite has presumably already been checked for correctness against the instructor implementation, or is axiomatically taken as correct.

Note that this model does not consider thoroughness as a criterion for including a student test suite. A test suite that is not thorough might not add much testing power, but it won't inaccurately mark an implementation as faulty. The other authors cited in section 2 include student tests at the granularity of a single test case. For simplicity sake, our work considers only entire test suites. Note that this does not change the flaws we identify, only the potential magnitude of our measurements.

A First Experiment. When we tried this technique on submissions for one of our assignments, the number of faulty implementations skyrocketed, from 28% of implementations identified as faulty by the instructor test suite alone, to 89% when student tests were also used. (We discuss details in section 6.) What happened?

This drastically different assessment outcome resulted from trusting tests that made assertions beyond the bounds of the specification. We illustrate the issue using two succinct example problems (that are simpler than those from our real data):

(1) Implement a function that computes a distance metric between two non-empty lists of values.
(2) Implement a function that transforms a list of numbers into a binary search tree. (The assignment does not specify in which branch duplicates should be placed.)

Each of these specifications, as given, admits multiple, functionally-distinguishable implementations. Respectively, the student implementations

(1) may do *anything* if either of the inputs are empty;
(2) may place equal values in either the left or right subtree.

The authors of these implementations are likely to write tests that assert whichever particular behavior they happened to choose. These tests, while correct with respect to the student's *own* implementation, are not appropriate tests of *all* implementations.

In general, such tests, which we term *over-zealous*, can exceed the bounds of the specification in two ways:

- *Be overly liberal in what they supply for inputs*; e.g., if the specification asks for a function that is only defined on non-empty lists, then a test that supplies the function with an empty list is over-zealous.
- *Be overly conservative in what they accept as outputs*; e.g., if the specification does not dictate to which side of the binary search tree duplicate elements should be placed, a test that assumes duplicates go to a particular side is over-zealous.

Pitfall of Over-Zealous Tests. Consider a student test suite whose over-zealous test cases *coincidentally* conform to the specific behavior of the instructor implementation. Since that suite will pass the instructor implementation, it will be labeled correct and then used to judge whether *other* student implementations are correct. Consequently, any other student implementations that behave differently (even if they satisfy the specification) will be marked faulty by that over-zealous test suite.

If two student test suites over-zealously test *different* aspects of the specification, and are both incorporated into the implementation assessment process, it can be virtually impossible for any implementation to be deemed correct. Both forms arose in our experiment, resulting in the dramatic increase in the percentage of implementations that were deemed faulty. This experiment illustrates why *eliminating over-zealous test cases from the implementation labeling process is crucial.*

Assessment Impact. If over-zealous tests are not eliminated, any conformant implementation that diverges even slightly from the instructor implementation may be wrongly judged as faulty. Furthermore, if this flawed labeling is used to assess student test suites, students will "fail" to identify these "faulty" implementations, and will be penalized for apparently having un-thorough test suites.

4.3 Algorithmic: Multiple Implementation (AlgMult)

Fundamentally, the problem created by over-zealous tests is one of over-fitting: while the specification describes a *space* of implementations, just *one* sample from that space (a single instructor implementation) is used to determine whether all other implementations conform to that specification. To mitigate this flaw, an instructor

can craft *multiple* correct implementations in a manner to be defined shortly. When assessing implementations, only test suites that are deemed correct against *all* of the instructor implementation are used in assessing other student implementations.

> **Model Summary:** Each student implementation is assessed against the instructor test suite as well as student test-suites that are correct relative to multiple instructor implementations.

In fig. 1, B's and D's test suites are checked against three instructor implementations. B's is consistent with all three, but D's appears to be over-zealous, failing implementation 2. Therefore, D's test suite is no longer considered, whereas B's test suite (whose checks denote passed instructor implementations) can be added to the pool of test suites for assessing A's implementation.

How should instructor implementations be different? Different instructor implementations should reflect different scenarios allowed by the specification (e.g., guarding against different kinds of over-zealous tests). In particular, different implementations might admit more inputs than the specification requires, or might produce outputs that are consistent with the specification in different ways. For example:

(1) If the specification only dictates how a function behaves on non-empty lists, then given an empty list, one instructor implementation might throw an exception while another returns an innocuous value.
(2) If the specification does not dictate to which side of a binary search tree duplicate elements should be placed, one instructor implementation might place them on the left, while another places them on the right.

Such implementations are *adversarial* in that they check for violations of the robustness principle.[1] A good set of adversarial implementations would be diverse enough that an over-zealous test suite would reject at least one of them. If this happens, over-zealous test suites would be ruled out before being used to assess other student implementations.

These restrictions against over-zealousness may appear to pose a high bar on student tests. Indeed they do, but the bar is not unattainable: in another experiment (section 6.1), the addition of adversarial instructor implementations reduced the fraction of student test suites trusted for assessing implementations from 79% to 35%. It is important to remember, however, that this high bar is for a test suite *to assess other student implementations*; it is not necessarily the bar we would use to grade the test suite itself.

We observe in passing that nothing in our description limits adversarial implementations to screening *student* tests. The fruits of labor to obtain additional tests by any means—from colleagues, by crowdsourcing, by the instructors themselves, etc.—should all pass through the same adversarial process before being used to assess student work. This burden is nevertheless worth bearing due to the problems created by the two more common methods of assessment (Axm and AlgSing).

[1]This principle is also known as *Postel's Law*, after Jon Postel's formulation of the principle in an early specification of TCP: "TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others." [28]

Assessment Impact. Through crafting multiple adversarial implementations, an instructor can defend against the risk of incorporating over-zealous tests. However, the consequences of misplacing trust in even a single over-zealous test are the same (and no less dire) than those described for the AlgSing method.

5 TESTING THE TESTER

A common flaw underlies the vulnerabilities of all three models: if an instructor does not adequately consider some aspect of the problem, their assessments of students may suffer. Taken individually, they provide neither a resolution nor a means to detect this flaw. While AlgMult partially defends against against the severe threat of mistrusting a student test, its defense relies on instructors' sufficient development of adversarial implementations. Instructors can avoid this risk entirely by using Axm instead of an algorithmic model to grade student implementations, but that leaves Axm's risk of penalizing students who detect bugs that the instructor failed to write tests for.

By leveraging *both* axiomatic and algorithmic labeling, an instructor can detect and resolve this flaw. Consider that for an assignment with an adequate set of adversarial implementations and an instructor test suite that is not out-matched by any valid student test, Axm and AlgMult must result in an *identical* correctness labeling of student implementations. If either of these conditions is false, there must exist an incorporated student's test that identifies some implementation as faulty that the instructor's test suite identified as correct. In this event, one of two possibilities must be true: (1) that the student test is, in fact, nonconformant, but there was not an adversarial implementation to identify it as such, or (2) that the student test is, in fact, conformant, and captures a behavior not explored by any test in the instructor's suite. The instructor should examine the test case in question, identify whether it is conformant, and either create an adversarial implementation that rules it out, or incorporate it into their test suite. In section 6, we apply this process to quantify the impact of these assessment flaws on a number of assignments in an introductory programming course.

6 EVALUATION ON COURSE DATA

To assess the extent to which these perils may *actually* impact the robustness of course assessment, we applied the models to reassess the submitted programs and test suites of students from a semester-long accelerated introduction to computer science course at a highly-selective private US university. The course is primarily taken by students with prior programming experience; students place into it based on a series of programming assignments over the summer. In one semester, the course covers most of the same material as the department's year-long introductory sequences (fundamentals of programming, data structures, core algorithms, and big-O algorithm analysis).

The course teaches functional programming (many students who place into it have prior experience with object-oriented programming), following techniques from the *How to Design Programs* [13] curriculum. Both this curriculum and the course emphasize testing. Students are required to submit test suites for every assignment. Test suites are graded for both correctness and thoroughness, and are weighted similarly to implementations in determining final

Assignment	% of Student Implementations Labled Faulty		
	Axm	AlgSing	AlgMult
DocDiff	34%	98%	46%
Nile	35%	97%	35%
Filesystem	22%	52%	26%
MapReduce	28%	89%	57%

Table 1: Percent of student implementations labeled faulty by each model.

Assignment	% of Student Test Suites Incorporated	
	AlgSing	AlgMult
DocDiff	90%	81%
Nile	28%	0%
Filesystem	94%	71%
MapReduce	79%	35%

Table 2: Percent of student test suites incorporated by each algorithmic model.

course grades. On some assignments, students submit test suites a few days before submitting implementations, receiving feedback on test-suite correctness in time to make modifications to their implementations and test suites.

For each assignment under study, we assessed student implementations under: (i) Axm, using the instructor test suite that was used during the semester to grade student implementations; (ii) AlgSing, using the instructor implementation that was used during the semester to grade the validity of student test suites; and then (iii) AlgMult, using the criterion specified in section 5 to develop adversarial implementations. We quantify the impact of Axm's and AlgSing's vulnerabilities by contrasting their outcomes to that of AlgMult.

Assignments Under Study. For the analysis for this paper, we selected four assignments that are quite different from each other and representative of the course overall:

- DocDiff, where 91 students implemented and tested programs computing a document similarity metric using a bag-of-words model [29].
- Nile, where 70 students implemented and tested a rudimentary recommendation system.
- Filesystem, where 76 students implemented and tested rudimentary Unix-style commands for traversing a (in-memory) file structure with mutually-dependent datatypes [13].
- MapReduce, where 38 pairs of students implemented and tested the essence of MapReduce [4] (implemented sequentially), and applied it to multiple problems. This included redoing some previous assignments (including Nile) in terms of the MapReduce paradigm, using their implementation.

We explored only four assignments because constructing multiple adversarial implementations is a potentially time-consuming process. For each assignment we constructed between two (for Nile) and seven (for MapReduce) adversarial implementations.

The differing number of students submitting for each assignment reflects students dropping the course (after DocDiff), then working in pairs (on MapReduce); the assignments are listed in the order in which they were assigned. On each assignment, there were a few (2-3, though 9 for Nile) submissions that were not included in the analysis (and are not reflected in the above counts): these assignments either had compile-time errors or threw run-time exceptions that we were not able to resolve with a few minutes of work.

6.1 Impact on Implementation Assessment

The models produced drastically different assessments of implementation correctness. Table 1 summarizes the percentage of student implementations that were deemed faulty under each of the three models (the MapReduce data were mentioned in section 4). Very few implementations are deemed faulty under Axm, the majority are deemed faulty under AlgSing, and AlgMult lies in between (that the AlgMult percentages are no smaller than those for Axm matches our expectation based on their definitions).

With the Axm model, we noted that an insufficiently thorough instructor test suite may fail to detect all faulty implementations. We assumed that our test suites for these assignments were thorough, but had not validated this belief. Contrasting the first and third columns of table 1, we find that a substantial proportion of students who were notified that their implementations were correct *actually* had faults in their submissions. These data confirm that there is significant room to improve the thoroughness of our tests: both DocDiff and MapReduce show notable differences in the percentage of faulty implementations flagged between Axm and AlgMult.

With the algorithmic models, instructors bolster their own test suites with student tests but, we noted, face the risk and consequences of inadvertently mis-trusting an over-zealous student test. In the case of AlgSing, instructors rely on just *one* known-correct implementation to filter out invalid tests. Contrasting the second and third columns of table 1, we find that an substantial proportion of the implementations marked faulty by AlgSing were, in fact, correct[2]. These data show that a single implementation was not sufficient for filtering student tests. Table 2 shows the percentage of students whose tests were incorporated by AlgSing and AlgMult. Contrasting its two columns, we find that AlgSing consistently over-trusted student tests.

6.2 Impact on Assessing Test Suites

Next, we explore the impact of these perils on test-suite assessment, working with our MapReduce data.

Methodology: The models in this paper classify implementations as correct or faulty. As we mention in section 4, we can then use this classification as a ground truth to assess the accuracy of student tests. We do this by applying each test suite to a collection of (assessed) implementations, and comparing the test suite's classification of their correctness against that provided by the model.

We perform this analysis on a collection of 53 MapReduce implementations. This collection contains all 38 student implementations,

[2]We do not report statistical significance, because the nature of these analyses introduce considerable nuance and difficulty in designing a statistical test. Regardless, for students, these differences have *personal* significance.

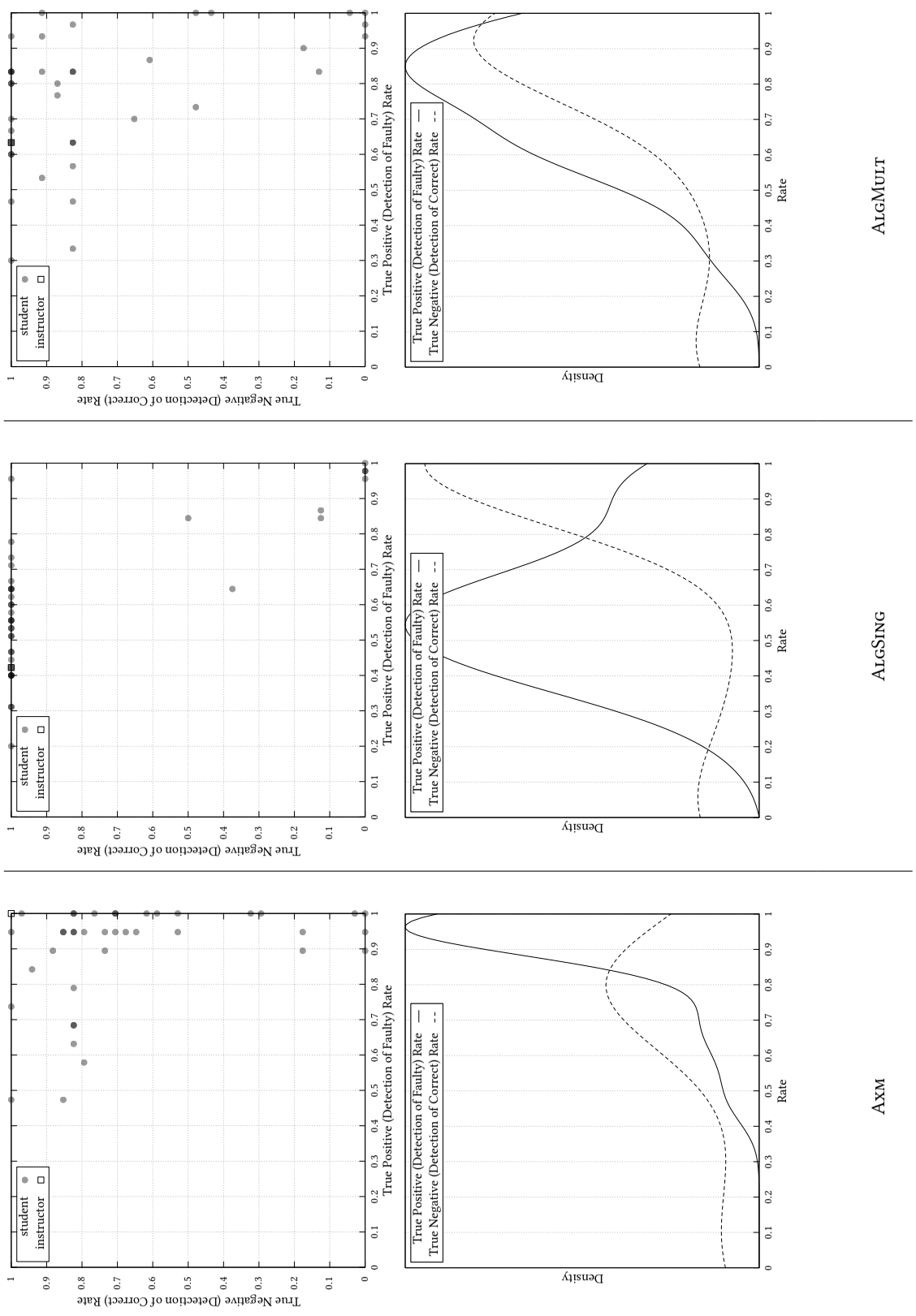

Figure 3: Assessment of test suites on MapReduce. In the top row, each dot represents a test suite; its location encodes its respective true-positive rate (the % of faulty implementations it accepted) and true-negative rate (the % of correct implementations that it rejected). Below, a kernel density estimation plot shows the relative commonality of true-positive and true-negative rates.

as well as seven (adversarial) correct and eight faulty specially-crafted implementations. These latter implementations were included to make sure that the corpus contained a handful of each kind of implementation (since we could not predict where the student implementations would fall).

We quantify the closeness of each student test suite's classification to the classification of the underlying models using the standard metrics of binary classifiers:

- *true-positive rate*, the fraction of faulty implementations that the test suite appropriately identifies as *faulty*;[3]
- *true-negative rate*, the fraction of correct implementations that the test suite appropriately identifies as *correct*.

Figure 3 depicts the resulting true-positive and true-negative rates of each test suite relative to the classifications produced by Axm, AlgSing, and AlgMult (one column each, respectively). These graphs illustrate, from the perspective of assessing student test suites, the drastically different outcomes that can arise depending on which model is used to label implementations.

Axm *Perils:* In the Axm model, a student that writes a test that identifies a bug missed by the instructor test suite is *penalized* for their thoroughness, as their test suite's judgment of correctness is observed as disagreeing with the judgment of the instructor's test suite. In the context of test suite assessment, this is reflected as a decrease in true-negative rate.

This pathology significantly impacted the outcomes of our test suite assessments performed atop Axm. The gaps in our test suites were accessible enough for *many* students to find (even though we had refined these test suites over several years). On DocDiff, Filesystem, and MapReduce, respectively, 63%, 9% and 29% of students identified at least one faulty implementation that instructor test suite missed. (Only on Nile did *no* student test more cleverly than the instructor had.) A contrast of the Axm and AlgMult columns of fig. 3 bears this pathology out in the context of test suite assessment. The true-negative density curve of Axm is shifted slightly to the left of that of AlgMult, indicating that Axm assessed students as having lower true-negative rates than AlgMult did. Furthermore, we note that Axm penalizes equally students who write invalid tests and students who test more cleverly than the instuctor. Thus, an instructor using Axm might incorrectly conclude that their best students failed to understand the problem specification.

AlgSing *Perils:* In theory, AlgSing and AlgMult (which acknowledge that students may find bugs the instructor does not) remedy this pathology. However, as discussed in section 6.1, incorrectly incorporating student tests can easily give rise to a catastrophically inaccurate assessment of implementations, which in turn leads to inaccurate assessment of test suites. Under AlgSing, most students have very high true-negative rates and very low true-positive rates. This came about in part because so few implementations were labeled correct by AlgMult (see the middle column of table 1). Thus, there is much less nuance in the true-negative rates, as reflected in the horizontal bands of points in the scatter plot.

[3]While associating "positive" with "faulty" may seem backwards, the goal of thoroughness is to accurately identify faulty implementations.

6.3 Takeaway

The theoretical flaws of the standard models had real, substantial impacts on our assessments. Using the technique in section 5 to develop an AlgMult assessment, we identified and corrected numerous shortcomings in our grading artifacts. This process required close examination of each assignment statement, and we also encountered ways in which our assignments could be made clearer. Thus, in addition to improving our grading system with this process, we have improved the assignments themselves.

7 DISCUSSION

In an era of growing enrollments and on-line courses, it is essential to understand the nuances of automated assessment, especially since it seems to naturally fit some aspects of computing. In particular, this fit can mask worrisome weaknesses. With automated assessment widespread in everything from K-12 and tertiary courses to MOOCs to programming competitions to job placement sites and more, its foundations require greater scrutiny.

In this paper we look closely at automated assessment of programs and of their first-cousins, test suites. Through pure reasoning, we show that the standard models (sections 4.1 and 4.2) can suffer from significant measurement flaws. We present a new model of assessment (section 4.3) and a corrective technique that utilizes it (section 5). The results of section 6 validate all these claims in practice when assessing both implementation and test suite quality.

The problems we find in these models are disturbing in two ways. First, the flaws can be subtle, so instructors and students many never notice them. Indeed, as we have noted, in some cases the assessment results in students appearing to do better than their true performance. This may give students a false sense of confidence in their abilities. Second, it is not trivial to extrapolate from the feedback of these models to identifying a systemic flaw in students' work. Especially in massive or disconnected settings, it may be difficult to identify the problems we raise. The sheer volume of data available may blind some people to the true quality of the data.

On a personal note, we can relate how easy these flaws are to overlook. Like many others educators, we had used the two flawed methods for nearly two decades, growing increasingly dependent on them with growing class sizes (a widespread phenomenon in the US). The initial purpose of this study was simply to test the quality of student tests, in comparison to an earlier study by Edwards and Shams [10]. As we began to perform our measurements, we wondered how *stable* they were, and started to use different methods to evaluate stability. When we noticed wild fluctuations—which made our analyses highly unreliable—we began to investigate why small changes to the implementation set would have large effects, which led to unearthing the problems reported in this paper.

We therefore conclude with a salutary warning. While automated assessments are valuable and have their place, their use—as with any machine-generated artifact that draws on a large set of data—requires significant reflection. Happily, we demonstrate that the method of multiple adversarial implementations (section 4.3) avoids the pathologies we have found in automated assessment, enabling us to draw on a larger pool of inputs (namely, to consider student test suites and implementations as well), which in turn results in better evaluation of student implementations and test suites.

REFERENCES

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. ACM, New York, NY, USA, 153–160. http://doi.acm.org/10.1145/1869542.1869567

[2] Michael K. Bradshaw. 2015. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 488–493. http://doi.acm.org/10.1145/2676723.2677247

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. http://doi.acm.org/10.1145/351240.351266

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. http://doi.acm.org/10.1145/1327452.1327492

[5] The Pyret Project Developers. 2018. *The Pyret Programming Language*. Chapter 2.2. https://www.pyret.org/docs/latest/testing.html

[6] The Rust Project Developers. 2018. *The Rust Programming Language*. Chapter 11. https://doc.rust-lang.org/book/second-edition/ch11-00-testing.html

[7] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *Journal on Educational Resources in Computing* 3, 3, Article 1 (Sept. 2003). http://doi.acm.org/10.1145/1029994.1029995

[8] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* 3, 3, Article 1 (Sept. 2003). http://doi.acm.org/10.1145/1029994.1029995

[9] Stephen H. Edwards and Zalia Shams. 2014. Comparing Test Quality Measures for Assessing Student-written Tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 354–363. http://doi.acm.org/10.1145/2591062.2591164

[10] Stephen H. Edwards and Zalia Shams. 2014. Do Student Programmers All Tend to Write the Same Software Tests? In *ITiCSE*. ACM, New York, NY, USA, 171–176. http://doi.acm.org/10.1145/2591708.2591757

[11] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. 2012. Running Students' Software Tests Against Each Others' Code: New Life for an Old "Gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 221–226. http://doi.acm.org/10.1145/2157136.2157202

[12] John English. 2004. Automated Assessment of GUI Programs Using JEWL. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*. ACM, New York, NY, USA, 137–141. http://doi.acm.org/10.1145/1007996.1008033

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (second ed.). MIT Press.

[15] George E. Forsythe and Niklaus Wirth. 1965. Automatic Grading Programs. *Commun. ACM* 8, 5 (May 1965), 275–278. http://doi.acm.org/10.1145/364914.364937

[16] Eric Foxley, Omar Salman, and Zarina Shukur. 1997. The Automatic Assessment of Z Specifications. In *The Supplemental Proceedings of the Conference on Integrating Technology into Computer Science Education: Working Group Reports and Supplemental Proceedings (ITiCSE-WGR '97)*. ACM, New York, NY, USA, 129–131. http://doi.acm.org/10.1145/266057.266141

[17] Xiang Fu, Boris Peltsverger, Kai Qian, Lixin Tao, and Jigang Liu. 2008. APOGEE: Automated Project Grading and Instant Feedback System for Web Based Computing. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 77–81. http://doi.acm.org/10.1145/1352135.1352163

[18] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing Throughout the Curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*. ACM, New York, NY, USA, 271–275. http://doi.acm.org/10.1145/563340.563446

[19] J. B. Hext and J. W. Winings. 1969. An Automatic Grading Scheme for Simple Programming Exercises. *Commun. ACM* 12, 5 (May 1969), 272–275. http://doi.acm.org/10.1145/362946.362981

[20] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. http://doi.acm.org/10.1145/2568225.2568271

[21] Peter C. Isaacson and Terry A. Scott. 1989. Automating the Execution of Student Programs. *SIGCSE Bull.* 21, 2 (June 1989), 15–22. http://doi.acm.org/10.1145/65738.65741

[22] David Jackson. 2000. A Semi-automated Approach to Online Assessment. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education (ITiCSE '00)*. ACM, New York, NY, USA, 164–167. http://doi.acm.org/10.1145/343048.343160

[23] David Jackson and Michelle Usher. 1997. Grading Student Programs Using ASSYST. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97)*. ACM, New York, NY, USA, 335–339. http://doi.acm.org/10.1145/268084.268210

[24] David G. Kay, Terry Scott, Peter Isaacson, and Kenneth A. Reek. 1994. Automated Grading Assistance for Student Programs. In *Proceedings of the Twenty-fifth SIGCSE Symposium on Computer Science Education (SIGCSE '94)*. ACM, New York, NY, USA, 381–382. http://doi.acm.org/10.1145/191029.191184

[25] Will Marrero and Amber Settle. 2005. Testing First: Emphasizing Testing in Early Programming Courses. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, New York, NY, USA, 4–8. http://doi.acm.org/10.1145/1067445.1067451

[26] Sebastian Pape, Julian Flake, Andreas Beckmann, and Jan Jürjens. 2016. STAGE: A Software Tool for Automatic Grading of Testing Exercises: Case Study Paper. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 491–500. http://doi.acm.org/10.1145/2889160.2889203

[27] Joe Gibbs Politz, Shriram Krishnamurthi, and Kathi Fisler. 2014. In-flow Peer-review of Tests in Test-first Programming. In *ICER*. ACM, New York, NY, USA, 11–18. http://doi.acm.org/10.1145/2632320.2632347

[28] Jon Postel. 1980. *Transmission Control Protocol*. Internet-Draft. Internet Engineering Task Force. https://tools.ietf.org/html/rfc761

[29] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (Nov. 1975), 613–620. http://doi.acm.org/10.1145/361219.361220

[30] Zalia Shams and Stephen H. Edwards. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-written Software Tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 53–58. http://doi.acm.org/10.1145/2493394.2493402

[31] K. K. Sharma, Kunal Banerjee, and Chittaranjan Mandal. 2014. A Scheme for Automated Evaluation of Programming Assignments Using FSMD Based Equivalence Checking. In *Proceedings of the 6th IBM Collaborative Academia Research Exchange Conference (I-CARE) on I-CARE 2014 (I-CARE 2014)*. ACM, New York, NY, USA, Article 10, 4 pages. http://doi.acm.org/10.1145/2662117.2662127

[32] Joanna Smith, Joe Tessler, Elliot Kramer, and Calvin Lin. 2012. Using Peer Review to Teach Software Testing. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 93–98. http://doi.acm.org/10.1145/2361276.2361295

[33] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2017. An Automated System for Interactively Learning Software Testing. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 98–103. http://doi.acm.org/10.1145/3059009.3059022

[34] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. 2005. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. http://doi.acm.org/10.1145/1082983.1083149

[35] Matthew Thornton, Stephen H. Edwards, Roy P. Tan, and Manuel A. Pérez-Quiñones. 2008. Supporting Student-written Tests of Gui Programs. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 537–541. http://doi.acm.org/10.1145/1352135.1352316

[36] Urs von Matt. 1994. Kassandra: The Automatic Grading System. *SIGCUE Outlook* 22, 1 (Jan. 1994), 26–40. http://doi.acm.org/10.1145/182107.182101