Grading-Based Test Suite Augmentation

Jonathan Osei-Owusu*, Angello Astorga*, Liia Butler*, Tao Xie†, and Geoffrey Challen*
*Department of Computer Science, University of Illinois at Urbana-Champaign
{jo28, aastorg2, liiamb2, challen}@illinois.edu
†Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
taoxie@pku.edu.cn

***Abstract*—Enrollment in introductory programming (CS1) courses continues to surge and hundreds of CS1 students can produce thousands of submissions for a single problem, all requiring timely and accurate grading. One way that instructors can efficiently grade is to construct a custom instructor test suite that compares a student submission to a reference solution over randomly generated or hand-crafted inputs. However, such test suite is often insufficient, causing incorrect submissions to be marked as correct. To address this issue, we propose the Grasa (GRAding-based test Suite Augmentation) approach consisting of two techniques. Grasa first detects and clusters incorrect submissions by approximating their behavioral equivalence to each other. To augment the existing instructor test suite, Grasa generates a minimal set of additional tests that help detect the incorrect submissions. We evaluate our Grasa approach on a dataset of CS1 student submissions for three programming problems. Our preliminary results show that Grasa can effectively identify incorrect student submissions and minimally augment the instructor test suite.**

***Index Terms*—CS education; testing; clustering; test generation**

I. Introduction

Enrollment in introductory programming (CS1) courses continues to surge at remarkable rates [1]. As current instructional resources cannot keep pace with the rapidly growing demand, instructors must adapt or face a decline in CS1-course quality [1]. One key area in need of critical attention is grading. For a programming problem, hundreds of students can produce thousands of submissions over the period of a semester. Each of these submissions must be graded to provide feedback critical to student learning [2]. However, grading approaches reasonable for a small class (e.g., inspecting and grading each submission by hand) are no longer viable as the class size increases. Automated testing, while not exclusive to a large class, remains the most common approach to evaluate a large number of CS1 student submissions related to programming.

One way for instructors to grade efficiently is to construct a custom test suite that compares a submission to a reference solution over randomly generated or hand-crafted inputs. Such test suite is often insufficient, failing to include corner-case inputs for exposing faults in some incorrect submissions. As a result, incorrect submissions may be mistakenly marked as correct[1]. A test suite of insufficient quality not only may cause unfairness to students (e.g., students who do not meet the intended expectations of a problem receive the same grade as students who do), but also may cause instructors to miss valuable opportunities to gain insight about issues in students' learning. This shortcoming is exacerbated as the class size increases and subsequently the number of different ways that submissions are incorrect increases.

[1]Correct ones may also be marked as incorrect; however, in our experience such cases are usually reported by students.

To address test-suite insufficiency in CS education, we propose a new approach of conducting test suite augmentation, namely, the Grasa (**GRA**ding-based test **S**uite **A**ugmentation) approach, in the space of multiple (correct or incorrect) implementations of the same specification. The main objective of Grasa is to augment the given instructor test suite with a *minimal* set of generated tests that aim to detect a maximum number of incorrect submissions. The reasons for emphasizing "minimal" tests in Grasa's main objective are (1) minimizing the generated tests can alleviate the burden of instructors who wish to maintain control of assigning different point values per test (e.g., some tests are more important than others) while manually inspecting the generated tests; (2) minimizing the generated tests can reduce the runtime cost for executing these tests, especially when they are run against a large number of submissions.

To accomplish the main objective, Grasa first runs the given instructor test suite T_I on each submission against a reference solution R to identify and focus on only those submissions C determined by T_I to be correct. Then we conduct Grasa's two techniques: behavioral-equivalence approximation and equivalence-guided test generation.

In the technique of behavioral-equivalence approximation, two programs (e.g., submissions and R) are *approximately behaviorally equivalent* if they produce the same outputs on the sample inputs (in the input space) produced by a structural test generator such as Pex [3]. We group programs into a cluster when these programs are approximately behaviorally equivalent. The cluster including R is the cluster of correct submissions, and each of the remaining clusters is a cluster of incorrect submissions.

Then in the technique of equivalence-guided test generation, we again leverage Pex to generate a minimal set of tests T_{TG} that help detect the incorrect submissions detected by the technique of behavioral-equivalence approximation. Finally, we use the generated tests to augment the given instructor test suite.

In summary, this paper makes the following main contributions:

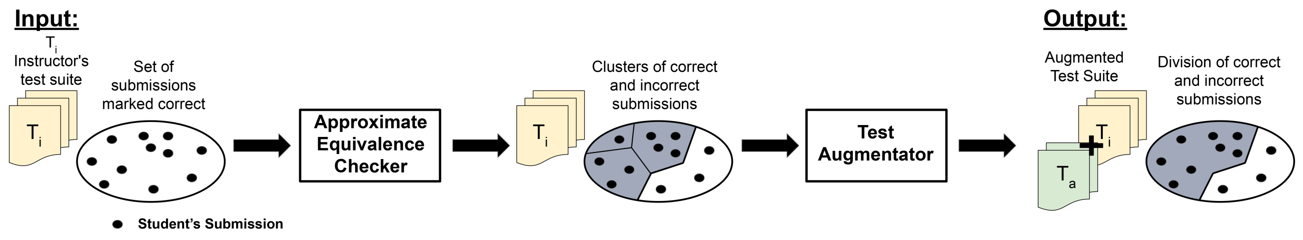
Input:
T_i Instructor's test suite
Set of submissions marked correct
T_i
● Student's Submission
Approximate Equivalence Checker
T_i
Clusters of correct and incorrect submissions
Test Augmentator
Output:
Augmented Test Suite
T_i
T_a
Division of correct and incorrect submissions

Fig. 1. Overview of the proposed Grasa approach.

- **GRAding-based test Suite Augmentation (Grasa)** for conducting test suite augmentation based on multiple (correct or incorrect) implementations of the same specification, with the target application domain as CS education.
- **Techniques** for identifying and clustering incorrect submissions and for augmenting the given instructor test suite with a minimal set of generated tests that help detect the incorrect submissions.
- **Preliminary Results** on submissions for three programming problems from a CS1 course spanning two semesters.

II. Approach

Our Grasa approach consists of two techniques: behavioral-equivalence approximation and equivalence-guided test generation, as shown in Figure 1.

A. Behavioral-Equivalence Approximation

Our technique of behavioral-equivalence approximation first checks the *behavioral equivalence* (in later parts of this section, we explain how to accomplish this checking in an approximate way) between each submission in the initial set of submissions C (determined by the instructor test suite T_I to be correct) with the reference solution R. Any submission non-equivalent to R is detected as an *incorrect submission*. Then all the submissions equivalent to R are put into the cluster of correct submissions.

For the remaining incorrect submissions, our technique further checks the behavioral equivalence between every two incorrect submissions. Two equivalent incorrect submissions are put into the same cluster of incorrect submissions (i.e., they are incorrect in the same way).

To check behavioral equivalence in an approximate way, we extend Paired Symbolic Execution (PSE) [4], [5]. PSE intends to generate test inputs for exposing different behaviors of two given programs sharing the same interface (e.g., the same method signature). Its key idea is to first construct a wrapper method to compare the outputs of the two programs given the same inputs, and then apply a structural test generator such as Pex [3] on this wrapper method to generate its argument values aiming to achieve high branch coverage of the wrapper method along with the methods directly or indirectly invoked by the wrapper method. Some generated argument values for the wrapper method may be able to cause different outputs of the two programs under comparison.

```
public static void WrapperForPairedSymbolicExecution(args) {
   assert(IsEqual(prog1(args), prog2(args));
}
```

Fig. 2. Simplified wrapper constructed for paired symbolic execution.

Figure 2 shows a simplified version of a wrapper method. In this simplified version, we assume that both programs are static methods, and their method arguments, denoted as `args`, are either primitive-type arguments or immutable objects. `assert` with the boolean argument is a typical assertion method (e.g., one commonly used in a unit testing framework). `isEqual` is a method for checking the equivalence of two values specified in its two arguments. When the two values are of non-primitive type, `isEqual` can be implemented as checking the object-state equivalence [6] based on its custom `equals` method.

When the method arguments `args` include mutable objects, the simplified wrapper method shown in Figure 2 is extended to declare two arguments denoted as `argProg1` and `argProg2` for each mutable argument of the programs. Then the beginning of the wrapper method body invokes an assumption method: `assume(isEqual(argProg1, argProg2))`. If any generated argument values for the wrapper method cause the boolean argument of `assume` to be false, the generated argument values are automatically discarded (i.e., classified as invalid).

When the programs under comparison are non-static methods (thus having receiver objects, being mutable), similarly the simplified wrapper method shown in Figure 2 declares two arguments denoted as `receiverProg1` and `receiverProg2`, and the beginning of the wrapper method body invokes `assume(isEqual(receiverProg1, receiverProg2))`.

Note that additionally, the end of the wrapper method body invokes the assertion method for asserting the equivalence of two updated receiver objects (for non-static methods under comparison) or two updated mutable argument objects (for methods under comparison including a mutable argument).

After applying a structural test generator such as Pex [3] on the wrapper method to generate tests (i.e., argument values for the wrapper method), we say that two programs are *approximately behaviorally equivalent*, in short as *equivalent* for simplicity, when all of the following three conditions are satisfied: (1) there is no generated test for causing violation of any assertion synthesized by us in the wrapper method body; (2) if the program inputs (e.g., receiver object and method argument values) derived from a generated test cause one program to throw an exception, then the (equivalent) program inputs derived from the same test also cause the other program to throw an exception of the same type; and (3) if the program inputs derived from a generated test cause one program to execute infinitely, then the (equivalent) program inputs derived from the same test also cause the other program to execute infinitely. To support Conditions 2 and 3 (which are heuristic by nature for behavioral-equivalence checking), we further extend the wrapper method with additional checking code.

B. Equivalence-Guided Test Generation

The technique of equivalence-guided test generation augments the given instructor test suite with a minimal set of tests that can detect the incorrect submissions (detected by the technique of behavioral-equivalence approximation), i.e., can detect behavioral inequivalence between the reference solution R and the incorrect submissions.

Note that a test that can detect behavioral inequivalence between R and a submission in a cluster can also detect behavioral inequivalence between R and all other submissions of the same cluster. Thus, to guide generation of minimal tests, our technique selects only one representative from each cluster of incorrect submissions, i.e., N representatives where N is the total number of clusters of incorrect submissions (resulted from the technique of behavioral-equivalence approximation).

Our technique works on iterations, each of which includes two steps. In Step 1, we conduct greedy test generation: generate a single test t that can detect behavioral inequivalence between R and the maximum number of representatives under consideration denoted as S_r (which initially include all N representatives). Our greedy test generation iteratively constructs a wrapper with a simplified one shown in Figure 3, and then applies a structural test generator such as Pex [3] on this wrapper method till the test generator generates an assertion-violating test. In Figure 3, `representative1`, ..., `representativeC` are all the representatives from S_r. G in Line 10 is a constant initially set as $|S_r|$ and decremented by 1 iteratively till an assertion-violating test is generated by the test generator.

In Step 2, from S_r we remove the representatives detected by t (generated in Step 1) to be incorrect (i.e., causing to cover the true branch of their corresponding `if` statements in the wrapper method body), and conduct greedy test generation again until S_r is empty.

Finally, all the tests t generated from all iterations are the minimal set of tests produced by our technique.

III. Preliminary Evaluation

A. Evaluation Setup

Our evaluation subjects include student submissions for three programming problems given in quizzes and exams in a CS1 course:

```
1  public static void WrapperForEquivGuidedTestGen(args){
2    int count = 0;
3    if !IsEqual(representative1(args), R(args))
4      count++;
5    if !IsEqual(representative2(args),  R(args))
6      count++;
7    ...
8    if !IsEqual(representativeC(args),  R(args))
9      count++;
10   assert(count < G);
11 }
```

Fig. 3. Simplified wrapper constructed for equivalence-guided test generation.

TABLE I
Preliminary Results

Problem	# Clusters	# Incorrect	# Tests	Min. # Tests
AddToEnd	5	10	3	2
Partitioner	3	4	3	2
Sort	0	0	0	0

- **`AddToEnd`**. A method that takes as input an integer and appends it to the end of a singly-linked list. This method is defined in a singly-linked list class (with member variables defined by the instructor) where students need only to fill in the `AddToEnd` method body.
- **`Partitioner`**. A method that takes as input an array of integers. The method returns an index such that the element to this index's left (right) is strictly less (greater) than or equal to the element stored at the indexed location.
- **`Sort`**. A method that takes as input an array of integers and returns the array sorted in the ascending order.

All of the student submissions are originally written in Java. We take the submissions classified as correct by the instructor test suite and translate them to C# via the Sharpen tool [7]. This translation is done to make the student submissions compatible with Pex [3], the used test generator[2]. In addition, we write factory methods to help Pex create complex objects as inputs (e.g., `AddToEnd`'s singly-linked lists) and `IsEqual` methods that help determine whether complex data structures are equivalent. For `AddToEnd`, there are totally 3,985 submissions, from which 308 are marked correct and 302 are used in our evaluation; we omit 6 submissions that cause translation failures. For `Partitioner`, there are totally 1870 submissions, from which 174 are marked correct and analyzed. For `Sort`, there are totally 4481 submissions, from which 242 are marked correct and analyzed.

B. Preliminary Results

Table I shows the results of applying our Grasa approach on the evaluation subjects. In two out of the three programming problems, Grasa detects incorrect submissions that are mistakenly classified by the existing instructor test suite as correct. In particular, for `AddToEnd` and `Partitioner`, Grasa detects 10 and 4 incorrect submissions, respectively. Grasa forms 5 and

[2]We use Pex because of its effectiveness in previous related work [5]. However, alternative structural test generators could also be used.

3 clusters for these incorrect submissions of `AddToEnd` and `Partitioner`, respectively.

In Table I, "# Tests" denotes the total number of incorrectness-exposing (IE) tests produced across all clusters, one IE test per cluster (when the same test is generated for multiple clusters, we count it only once). "# Min. Tests" denotes the number of minimal IE tests produced by Grasa. The results show that for both `AddToEnd` and `Partitioner`, "# Tests" as 3 is reduced to "# Min. Tests" as 2. We conduct additional experimentation to confirm that 2 IE tests are indeed the minimal tests for exposing the incorrect submissions.

An example IE test produced by Grasa to augment `AddToEnd`'s instructor test suite appends an integer to the end of a list with an initial size of 1. In the instructor test suite before augmentation, 32 instructor-constructed lists used for testing are randomly generated with constrained size limits between 1 and 32. However, randomly generating a list of size 1 is not guaranteed. 7 of the 10 Grasa-identified incorrect submissions exhibit undesirable behavior in this corner case.

Note that because the instructor test suites (including random-test generation) constructed by the instructor for the three programming problems are of quite high quality already, we expect the extent of benefits brought by Grasa to be much higher for general cases.

IV. Related Work

Existing work on test suite augmentation [8], [9], [10] focuses on generating additional tests that cover the (typically small) changes from one version of a program to its next version. However, the existing work does not address the case (focused by our approach) where a test suite can be used to assess the correctness of multiple implementations with respect to the same specification, e.g., student submissions and reference solution. In particular, the existing work often cannot handle changes across these multiple implementations, given that these changes are typically much more substantial than those changes across nearby versions of a program.

Due to rapid growth in enrollment of CS courses, various approaches help instructors understand a large number of student submissions and help enable other analyses such as automated repair and feedback generation. For example, Gulwani et al. [11] cluster correct programs based on their runtime control flow and variable-value equivalence throughout program execution in order to generate repairs for incorrect programs. Head et al. [12] cluster programs based on learned code transformations for representing a fix. In order to learn a transformation, their approach requires the existence of a pair of incorrect and correct programs being sufficiently similar, whereas our approach does not have this requirement. Singh et al. [13] formulate feedback generation as a synthesis problem; our future work plans to use synthesis-based approaches [14] to learn preconditions as feedback for summarizing incorrect behaviors of incorrect student submissions.

V. Conclusion

To address the insufficiency of an instructor test suite for grading student submissions, in this paper, we have presented the Grasa (**GRA**ding-based test **S**uite **A**ugmentation) approach that first detects and clusters incorrect submissions and then generates a minimal set of additional incorrectness-exposing tests to augment the existing test suite. Our preliminary evaluation results on three programming problems demonstrate the effectiveness of Grasa.

In our approach, checking behavioral equivalence is approximated with a structural test generator, whose checking is incomplete by nature. Our preliminary results show that our test generation can still find additional incorrect submissions not detected by the existing instructor test suite, which is already of quite high quality with random test generation.

In addition, the provided reference solution can be incomplete specification of the problem's desired behaviors. If the reference solution captures only one of multiple possible desired behaviors, then our approach would misclassify correct submissions as incorrect, but such misclassification cases can often be caught by either students or instructors. In addition, our approach can be easily extended to support multiple reference solutions with different desired behaviors: a submission is determined as correct when its behavior is equivalent to one of the multiple reference solutions.

Acknowledgment

We thank Jacob Laurel, Wing Lam, August Shi, and Nicholas Strole for their feedback, and Vijayendra Japtap for his help on our earlier work. This work was supported in part by NSF under grant no. CNS-1564274, CCF-1816615, the NSF of China under Grant No. 61529201, a grant from Futurewei, and the GEM fellowship.

References

[1] T. Camp, W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker, and S. Zweben, "Generation CS: The growth of computer science," *ACM Inroads*, 2017.
[2] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman, *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Jossey-Bass, 2010.
[3] N. Tillmann and J. de Halleux, "Pex–white box test generation for .NET," in *TAP 2008*.
[4] K. Taneja and T. Xie, "DiffGen: Automated regression unit-test generation," in *ASE 2008*.
[5] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *ICSE 2016 SEET*.
[6] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *ASE 2004*.
[7] Sharpen. https://github.com/mono/sharpen.
[8] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE 2008*.
[9] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided path exploration for efficient regression test generation," in *ISSTA 2011*.
[10] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *PLDI 2011*.
[11] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *PLDI 2018*.
[12] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in *L@S 2017*.
[13] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI 2013*.
[14] A. Astorga, P. Madhusudan, S. Saha, S. Wang, and T. Xie, "Learning stateful preconditions modulo a test generator," in *PLDI 2019*.