Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics

Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University, University Park
{xvl5190, swz175, pxw172, dwu}@ist.psu.edu

***Abstract*—Programming assignment grading can be time-consuming and error-prone if done manually. Existing tools generate feedback with failing test cases. However, this method is inefficient and the results are incomplete. In this paper, we present AUTOGRADER, a tool that automatically determines the correctness of programming assignments and provides counterexamples given a single reference implementation of the problem. Instead of counting the passed tests, our tool searches for semantically different execution paths between a student's submission and the reference implementation. If such a difference is found, the submission is deemed incorrect; otherwise, it is judged to be a correct solution. We use weakest preconditions and symbolic execution to capture the semantics of execution paths and detect potential path differences. AUTOGRADER is the first automated grading tool that relies on program semantics and generates feedback with counterexamples based on path deviations. It also reduces human efforts in writing test cases and makes the grading more complete. We implement AUTOGRADER and test its effectiveness and performance with real-world programming problems and student submissions collected from an online programming site. Our experiment reveals that there are no false negatives using our proposed method and we detected 11 errors of online platform judges.**

***Index Terms*—automatic grader, programming assignments, weakest precondition, equivalence checking**

I. INTRODUCTION

Providing prompt feedback on programming problem submissions encourages students to adaptively learn coding skills and allows instructors to collect real-time information about how their teaching goals are being met. In this context, manual grading can be both time-consuming and error-prone, especially when instructors are teaching large-sized classes. With the popularity of Massive Open Online Courses (MOOCs), (e.g., the various programming courses on the edX platform, which has more than 150,000 students enrolled worldwide [1]), prompt grading of all students submissions has become more challenging. Many online courses adopt peer feedback [2], but this type of feedback is not as responsive as desired. Students have to wait for hours to receive the feedback and instructors have only limited knowledge of these unstructured peer evaluation processes.

Automated grading and feedback generation have been actively researched in the past decade. A survey by Ihantola et al. [3] reviewed various tools for the automatic assessment of programming exercises. Existing tools generate feedback with failed test cases where test inputs can be either automatically generated [4], [5] or selected from a collection of representative test suites provided by instructors [6]–[9]. However, there are two critical issues affecting test-based grading. First, it can be costly to construct high-quality test cases. In fact, providing a complete test suite for a programming problem may not be practically feasible because of limited resources. After scanning the online judging platform forums [10]–[12], we noticed numerous user complaints about a lack of test cases for certain programs, which can lead to incorrect judging. This potentially allows students to overuse the grading system with incomplete test suites by collecting failed test cases with many submission attempts. Another problem with test-driven grading is that some tools generate feedback using formal methods [13]–[15] if instructors have reference solutions to a programming problem. An error model is utilized to provide more in-depth and comprehensive feedback. However, it is arguable that writing a complete set of reference specifications is less expensive than constructing test cases, especially when the problem can be solved in numerous ways. Error models also require additional effort to devise. Most importantly, to the best of our knowledge, the overall accuracy of these tools is quite limited, hindering their deployment in the real world.

In this paper, we propose AUTOGRADER, a tool that automatically provides real-time judgments with counterexamples for programming exercises in introductory programming courses. The solution is based on program analysis, and more specifically, the technique of differential semantic analysis. With a *single* correct implementation as the reference solution, AUTOGRADER can "search" for the differences between the execution traces of a student's submission and the reference solution. If one or more dissimilarities are found, the submission is decided as incorrect; otherwise, the submission is decided as correct. We have designed our method to find the inputs that will cause two programs to behave differently, by having either different output states or semantically different execution paths. Although it is simple to compare output states, proving the equivalence of execution paths is quite challenging. To tackle this problem, we leverage the idea of path deviation [16], [17] with improvements tailored to the autograding problem. To be more precise, we compare two execution paths by looking at their input spaces. The execution path should have one-to-one trace correspondence for satisfiable inputs. If a point in the symmetric difference of the two input spaces that can lead to other semantically

different paths is found, the compared programs are proven inequivalent, thus making the submission incorrect.

We implement AUTOGRADER with symbolic execution and weakest precondition. We also develop an *Equivalence Checker* to distinguish true semantic deviations from "syntactic" deviations. Our method is the first to apply program traces for the automated grading of programming problems with a *single* correct reference implementation. In this regards, our solution has a number of advantages. First, compared with purely test-based methods, our method significantly alleviates the workload of instructors, saving them from constructing test cases. Second, our method is based on rigorous program semantics, verification conditions (weakest precondition), symbolic execution, and constraint solving. Our method is also sound regardless of the limitations of symbolic execution and constraint solving. In practice, we achieved no false negatives in autograding this study's programming assignments; and, our tool identified 11 false negative errors when using a real-world test-based grading method. In this way, we reveal the weakness of incomplete test suites that are often used in the online grading of programming assignments.

II. OVERVIEW

A. Problem Statement

The goal of our work is to automatically decide the correctness of student programming assignments according to a reference implementation. To be more specific, given a reference implementation R and a student submission S, our goal is to decide whether S behaves the same as R. That is, we seek to provide a *yes* or *no* answer to the following question: Are S and R semantically equivalent?

Suppose a programming assignment requires students to write a `max3` function that returns the max value of three integer inputs. The instructor provides a reference implementation (Fig. 1a), and we receive five student submissions (Fig. 1b–1f), where *Submission* 1 and *Submission* 2 are correct implementations and the rest are incorrect ones. While the correct submissions are semantically equivalent to the reference implementation, the incorrect programs neglect boundary conditions. Given the input $a = 5, b = 5, c = 3$, *Submission* 3 and *Submission* 4 will return 3. Given the input $a = 5, b = 3, c = 3$, *Submission* 5 will return 3. Both of these outcomes are incorrect. We expect AUTOGRADER, which relies on the correct reference implementation, to automatically decide that *Submission* 1 and *Submission* 2 are correct and that the other three are incorrect.

B. Basic Idea

In general, a program can be characterized with respect to three aspects: *program inputs*, *program outputs*, and *program execution paths*. Our idea is to efficiently search for semantic differences between a student's submission and the reference implementation by detecting *Path Deviation* [16]–[18]. To be more specific, two programs, a reference R and a submission S, should follow a certain path given the same input i_1. When given a different input i_2, the two programs should either both follow the original path or both execute a new path (a semantically different one). Otherwise, we can say that there is a deviation; that is, with the new input, one program follows the original path while the other program follows a semantically different path. Hence, our goal becomes finding such an input that causes the new execution to deviate from the original path, thereby revealing the incorrectness of the submission program.

To find such an input, our methodology relies on a form of differential program analysis that performs symbolic execution to precisely characterize program behaviors by collecting the path conditions and computing the weakest preconditions. We use the accumulated path conditions, denoted by F, to represent the trace. Therefore, we have two formulas, F_r for the trace in the reference implementation and F_s for the trace in the student submission.

At a high level, it is easy to find an input i such that $R(i) \equiv S(i)$, where $R(i)$ and $S(i)$ are the output states or values, assuming students only make mistakes in corner cases but process the inputs with correct logic in most cases. Assume we run the reference program R and the student submission program S with the input i, and doing so yields two traces, T_r and T_s. We then build two formulas, F_r and F_s, representing the two traces' semantics (with the path condition and weakest precondition). That is, if $F_r(i)$ holds for input i, the program execution will follow trace T_r. The submission trace functions similarly.

Then the key step is to find another input x that causes one of the two traces to deviate semantically while the other stays on the original trace. That is, either $F_r(x)$ or $F_s(x)$ holds, but not both. Therefore, we have

$$\exists x.(F_s(x) \wedge \neg F_r(x)) \vee (\neg F_s(x) \wedge F_r(x)). \tag{1}$$

This formula reveals the semantic difference between the two traces. The recorded symbolic formulas are fed to a constraint solver, and we check the satisfiability of $(F_s \wedge \neg F_r) \vee (\neg F_s \wedge F_r)$ to reveal path deviations. The satisfiability of such a formula signifies the existence of inputs that causes the corresponding execution traces to deviate. That is, satisfiable solutions represent program inputs that can cause the execution traces between the reference and student submission to deviate, thus revealing an incorrect submission. Furthermore, each satisfiable solution serves as a concrete example to trigger the deviation and can hence be directly used for debugging purposes. In another case, if there is no satisfiable solution for the formula, we try another trace until all the paths in the program are covered.

However, the "syntactic" path deviation does not necessarily lead to a true semantic deviation due to "trace splitting". Trace splitting can typically lead to false positives at this stage. To make sure the new input x triggers a real deviation, we undertake an equivalence checking of the deviating path and the original path (see details in §III-E). More specifically, we check whether there are equivalent paths for the new execution

```
1  int max3(int a, int b, int c) {
2    if (a>=b && a>=c)
3      return a;
4    else if (b>=a && b>=c)
5      return b;
6    else
7      return c;
8  }
```

(a) Reference

```
1  int max3(int a, int b, int c) {
2    if (a>=b)
3        if (a>=c)
4            return a;
5        else
6            return c;
7    else
8        if (b>=c)
9            return b;
10       else
11           return c;
12 }
```

(b) Submission 1 (Correct)

```
1  int max3(int a, int b, int c) {
2    if (a>b)
3        if (a>c)
4            return a;
5        else
6            return c;
7    else
8        if (b>c)
9            return b;
10       else
11           return c;
12 }
```

(c) Submission 2 (Correct)

```
1  int max3(int a, int b, int c) {
2    if (a>b && a>c)
3        return a;
4    else if (b>a && b>c)
5        return b;
6    else
7        return c;
8  }
```

(d) Submission 3 (Incorrect)

```
1  int max3(int a, int b, int c) {
2    if (a>b && b<c)
3        return a;
4    else if (b>a && a>c)
5        return b;
6    else
7        return c;
8  }
```

(e) Submission 4 (Incorrect)

```
1  int max3(int a, int b, int c) {
2    if (a>b && b>c)
3        return a;
4    else if (b>c)
5        return b;
6    else
7        return c;
8  }
```

(f) Submission 5 (Incorrect)

Fig. 1. Reference implementation and student submissions

path with the same inputs in the reference implementation by solving the formula

$$(O_r \neq O_d) \wedge (F_d \wedge F_r), \tag{2}$$

where O is the output value and the subscripts r and d represent the *reference trace* and the *deviation trace*, respectively. Basically, we try to answer the question: *Can we find an input at the intersection of the input spaces of the two traces that leads to different output values?* If the constraint solver reports "unsatisfiable", this means that the detected deviating path is a condition that can be merged into an existing path. Our tool reports a false deviation for *Submission* 2 by solving the first formula. However, if the constraint solver reports "satisfiable" for the second formula, the case of unmerged paths is canceled and we report a true deviation.

We note that our method is very efficient in terms of finding deviations. *If there is a semantic path deviation associated with the original execution traces, our method is able to find the deviation in one step, with the help of symbolic execution, weakest precondition, and constraint solving.*

C. A Running Example

Fig. 1a presents a sample reference implementation. Two conditional statements exist in this program, leading to three execution paths. If a conditional statement holds, the execution path will be chosen directly and corresponding instructions will be executed. For instance, in the reference implementation, if the first conditional statement on *line 2* holds, the formula for this execution path will contain $(a >= b \wedge a >= c)$ and the corresponding output will be `a`. Similarly, the accumulated path condition will contain $\neg(a >= b \wedge a >= c)$ if the conditional statement does not hold.

Fig. 2 shows the workflow of AUTOGRADER. We present how AUTOGRADER works using a running example as shown in Fig. 1. To grade a programming submission, we first generate a non-trivial program input using a white-box fuzzer [19]. With the generated input, in this case $a = 5, b = 3, c = 2$, we record the program execution traces for the five programs. We then construct a constraint given the formulas of the two execution traces, describing potential path deviations. The constructed constraint is fed to a constraint solver to check whether a path deviation truly exists. Details of each step are given in §III.

In the running example, we can quickly decide that *Submission* 4 is not correct because of the incorrect output. While the correct output from the reference implementation is 5, *Submission* 4 returns the value 3. For other submissions, we compute the following formulas for each trace:

$$F_r = ((a \geq b) \wedge (a \geq c))$$
$$F_{s_1} = ((a \geq b) \wedge (a \geq c)) \quad F_{s_2} = ((a > b) \wedge (a > c))$$
$$F_{s_3} = ((a > b) \wedge (a > c)) \quad F_{s_5} = ((a > b \wedge b > c))$$

To detect any deviations, we must fundamentally solve the formula $(F_s \wedge \neg F_r) \vee (\neg F_r \wedge F_s)$ for each of the submissions. Table I shows the results of the satisfiability check for this example. It is easy to see that in *Submission* 1, F_r and F_{s_1} are the same. Our tool confirms this by reporting "unsatisfiable" for constraint $(F_{s_1} \wedge \neg F_r) \vee (\neg F_r \wedge F_{s_1})$. For the rest of the submissions, the constraint solver finds inputs that cause the execution paths in the reference implementation and submissions to deviate. These deviations are documented in Table I.

While the newly generated input will trigger the same path in the reference as the original input (`line 2`), we may notice that they also trigger deviations in *Submission* 2 (`line 8`), *Submission* 3 (`line 6`) and *Submission* 5 (`line 4`). These deviations can be caused by syntactic "trace splitting". For example, although we can detect inputs (i.e., $a = 5, b = 5, c = 3$) for *Submission* 2 that cause the program trace to deviate, this is a false positive that we can neglect. This is because the detected input triggers the boundary condition $a = b$ on this continuous function. Both cases are correct if

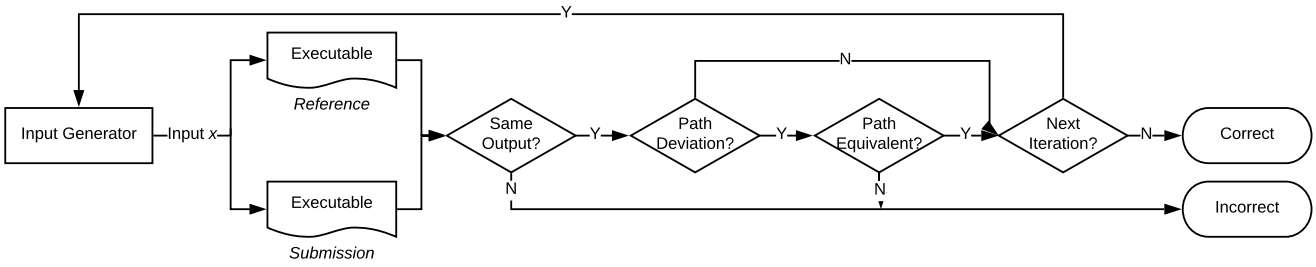

Fig. 2. The workflow of AUTOGRADER

we syntactically merge the boundary into the path $a > b$ or the path $a < b$. To confirm that there is a real path deviation, we perform equivalence checking of the deviating path and the original path on this trace. We solve Formula (2), and the satisfiable solutions show that there are inputs that hold both path constraints (F_d and F_r) and can lead to different outputs for the two paths. In other words, we confirm that the deviating path can truly lead to incorrect program outputs. Otherwise, if the constraint solver reports "unsatisfiable" for this constraint, we conclude that no input can lead to different program outputs along these two traces. For the cases in the example, we can confirm the deviations in *Submission* 3 and *Submission* 5, but the deviation in *Submission* 2 is ruled out.

If there is no path deviation reported for the two traces with a same input on the reference implementation and submission, our tool will ask the white-box fuzzer to generate another input on a new path until all paths in the reference implementation are covered or a time limit is reached. If there is no input that can lead to a path deviation or different program outputs, then it is safe to report the submission as correct. In the motivating example, our tool shows that *Submission* 1 and *Submission* 2 are correct as expected.

III. DESIGN

In this section, we present the design of AUTOGRADER. The workflow of AUTOGRADER is shown in Fig. 2. Given the reference implementation and one submission, we first compile these two programs into executable files. We then ask a white-box fuzzer to generate one program input and log the program execution trace by running the program executables with such an input (§III-A). The next step is to compare the execution outputs; different execution outputs indicate the incorrectness of the submission. If the outputs are identical, however, we undertake further analysis to identify potential path deviations along the trace (§III-C and §III-D). As previously discussed, path deviations may not lead to real semantic differences, and we further rule out false positives by checking path equivalence between the two traces (§III-E). Once a true positive (i.e., semantically different execution paths under the same input) is confirmed, we halt the analysis, grade the submission as incorrect, and output the counterexample. If no difference can be detected with the given input, we re-run the white-box fuzzer to yield new inputs and analyze further traces. A submission is concluded as correct once all the paths have been covered and no true positive deviations can be detected. We elaborate on each of these steps in the following sections.

A. Input Generation

White-box fuzzing ([20], [21]) is a commonly used technique to improve path coverage in software testing. Leveraging symbolic execution with constraint solving, white-box fuzzing efficiently generates test cases that can cover most of the program. In this research, we use the same technique to generate program inputs.

B. Trace Logging

Once an input is generated, we execute the program with that input and record the execution trace. We employ a dynamic program instrumentation tool to log the execution trace of the program.

A program execution trace contains a sequence of assembly instructions. In order to perform the symbolic execution, we first lift the low-level assembly representations into an analysis-friendly format. However, our tentative tests show that symbolic execution (see §III-C for details) can lead to a high performance penalty if every instruction on the trace is symbolically interpreted. Moreover, usually only a subset of the instructions on the trace is indeed relevant to program inputs. Thus, we perform taint analysis on the logged execution trace: Instructions that are irrelevant according to the data flow, given the program inputs, are ruled out in this step. After that, only the tainted instructions are kept for further analysis. Technical details regarding how we perform this analysis are discussed in §IV.

C. Symbolic Execution

As mentioned, our research aims at detecting inputs that can lead to path deviations. Recall that we seek to identify the *symmetric difference* (the symmetric difference of two traces defines the set of inputs that execute on either of the traces but not both) within the path conditions of the two execution traces over the same input space. The recognized differences are expected to reveal path deviations and even potential semantic divergences. Considering the input of this analysis as one concrete execution trace, this step aims to recover rich sets of information (e.g., input space over an execution path, path conditions) that represent the abstract semantics of the program. We solve the abstract semantics recovery problem with symbolic execution [22], [23].

TABLE I
PATH DEVIATION

#	Output	Formula (1)	Deviation	Deviation Example	Formula (2)	Equivalence	Decision
1	Correct	Unsat	No	-	-	-	Next Iteration
2	Correct	Sat	Yes	$a = 5, b = 5, c = 3$	Unsat	Yes	Next Iteration
3	Correct	Sat	Yes	$a = 5, b = 5, c = 3$	Sat	No	Incorrect
4	Incorrect	-	-	-	-	-	Incorrect
5	Correct	Sat	Yes	$a = 5, b = 3, c = 3$	Sat	No	Incorrect

By abstracting program inputs as symbols, symbolic execution interprets program traces and formulates each branch condition along the traces. Symbolic execution is considerably more precise than traditional data-flow analysis, and when the constraint solver finds a solution for the path deviation constraint, it naturally provides counterexamples that lead to variant branch selection, making it easier for users to debug their submissions.

In general, we perform symbolic execution along the trace and build a path constraint formula parameterized with the inputs. Our symbolic execution models each branch condition, and the path constraint formula is a conjunction of all the branch conditions. Given symbolic formulas from two execution traces, we build a path deviation constraint and attempt to find a satisfiable solution by invoking the constraint solver (see §III-D for details).

D. Path Deviation Identification

After constructing the path constraint formula for each trace, the next step is to identify inputs that yield path deviations. The workflow is shown in Fig. 3. Intuitively, given path constraints from two execution traces, inputs that can lead to path deviation should satisfy one constraint formula, while unsatisfying the other one. In other words, we would like to find the *symmetric difference* given the formulas from two paths. To this end, we propose the following constraint:

$$(F_s \wedge \neg F_r) \vee (\neg F_s \wedge F_r)$$

The above constraint contains formulas derived by analyzing the execution trace of the reference (F_r) as well as the submission (F_s). Constraint $F_s \wedge \neg F_r$ searches for inputs that satisfy the formula for the submission but not the reference (i.e., a path deviation in the reference), and constraint $\neg F_s \wedge F_r$ finds path deviations in the submission trace. In general, satisfiable solutions for the above constraint indicate the existence of path deviation, and whenever there is no satisfiable solution for the above constraint, we are assured that the given trace is free from path deviation.

Example. Consider the following two programs: one checks condition $x > 0$, while the other checks $x > 1$:

```
f(x) = if (x > 0) then y = 2 else y = 3
g(x) = if (x > 1) then y = 2 else y = 3
```

Given the input 0, symbolic execution yields constraints $\neg(x > 0) \wedge (y = 3)$ and $\neg(x > 1) \wedge (y = 3)$ for the two programs, respectively. Hence, the path deviation constraint for these two traces is:

$$(\neg(x > 0) \wedge (x > 1)) \vee ((x > 0) \wedge \neg(x > 1))$$

Identifying satisfiable solutions for the above constraint is a typical satisfiability modulo theory (SMT) problem, and we employ SMT solvers to solve the constraint. We note that if the constraint is solvable, the SMT solver provides a counterexample (i.e., the "satisfiable" solution) that can cause the path deviation. With the original input ($a = 5, b = 3, c = 2$), our motivating example *Submission* 2 (Fig. 1c) yields a path deviation constraint $((a \geq b) \wedge (a \geq c)) \wedge \neg((a > b) \wedge (a > c)) \vee \neg((a \geq b) \wedge (a \geq c) \wedge ((a > b) \wedge (a > c))$, and the SMT solver reports "satisfiable" together with a concrete counterexample ($a = 5, b = 5, c = 3$). Obviously, this counterexample can truly cause a path deviation in the trace of *Submission* 2.

E. Path Equivalence Check

To compare a submission with the reference implementation, our technique calculates path deviations given the execution traces of two implementations (§III-D). However, as previously mentioned, the existence of path deviations may not actually lead to incorrect submissions. Suppose analyzing path R_1 and S_1 reveals a path deviation S_2 in the submission program. Then path R_1 may actually be equivalent to S_2 under the input space as the conjunction of S_2 and R_1. Thus, this yields a false positive of the deviation detection.

Our study shows that it is very common to find in student submissions that students prefer to solve a problem case-by-case. For example, the analysis of our motivating examples (Fig. 1) shows that given input $a = 5, b = 5, c = 3$, *Submission* 2 (Fig. 1c) and *Submission* 3 (Fig. 1d) both have path deviations compared to the reference. However, further study of these two submissions shows that the deviating path in *Submission* 2 actually yields the same output as the reference, while only *Submission* 3 has a truly incorrect output.

To tackle this issue, we propose checking the *path equivalence* between two suspicious paths in different programs. The workflow is illustrated in Fig. 4. To illustrate this more clearly, suppose our path deviation analysis identifies one deviation in the submission program. Then our goal in this step is to compare the semantics of the deviating path P_d in the submission with the corresponding path P_r in the reference, given one input x_d that causes the deviation.

Two paths from different programs are considered to have "path equivalence" if their semantics are always identical. Hence, we check whether it is possible to identify an input from the conjunction of two path constraints that would yield different program outputs. Given inputs from the conjunction of two path constraints, two programs must execute the desired

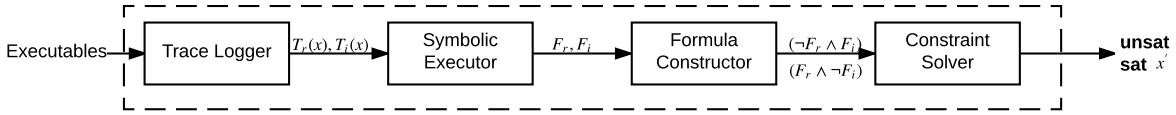

Fig. 3. Deviation identification

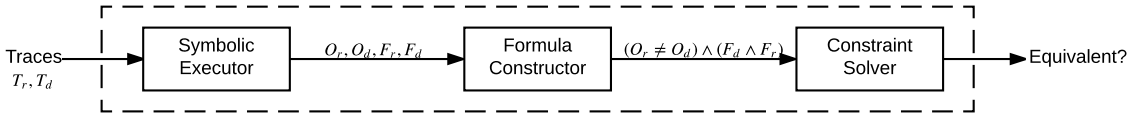

Fig. 4. Path equivalence checking

path and reach the observed program points, at which time we check whether the outputs are different. Thus, we propose the following constraint:

$$(O_r \neq O_d) \wedge (F_d \wedge F_r)$$

O represents the symbolic outputs of the two programs parameterized with the input variables, and F represent the path constraints of the two paths. In general, satisfiable solutions for the above constraint show that there is an input that satisfies both path constraints (F_d and F_r) and can yield different outputs for the two paths. In other words, we confirm the deviating path (P_d) can truly lead to incorrect program outputs (i.e., a true positive). Conversely, if the constraint solver reports "unsatisfiable" for this constraint, we conclude that no input can lead to different program outputs along these two traces, thus ruling out a false positive in our path deviation analysis.

In *Submission* 2 and *Submission* 3 of our motivating example, given the deviation paths detected with input $a = 5, b = 5, c = 3$, the corresponding symbolic outputs are:

$$O_r = a, O_{d_2} = b, O_{d_3} = c.$$

And the path constraint for each trace is:

$$F_r = ((a \geq b) \wedge (a \geq c))$$
$$F_{d_2} = (\neg(a < b) \wedge (b > c))$$
$$F_{d_3} = (\neg((a > b) \wedge (a > c)) \wedge \neg((b > a) \wedge (b > c))).$$

After checking path equivalence constraints, the constraint solver reports "unsatisfiable" given the constraint derived from F_r and F_{d_2}, though it identifies satisfiable solutions for the other case. Thus, we can confirm that *Submission* 3 is a true positive (i.e., incorrect submission) and output the feedback with the generated input $a = 5, b = 5, c = 3$ as a counterexample. We can then re-launch the symbolic execution (§III-C) for a new trace of *Submission* 2 and the reference.

IV. Implementation

AutoGrader is written in Python and has over 1,000 lines of core implementation. We employ an open-source white-box fuzzer (Pathgrind [19]) to generate test inputs. To make the most of this engine, we rewrite some of the test programming assignments to fit the input format.

The program execution trace is generated by Pin [24], a widely used dynamic binary instrumentation tool. Pin provides the infrastructure to intercept and instrument the execution of a binary. During execution, Pin inserts the instrumentation code into the original code and recompiles the output with a Just-In-Time (JIT) compiler. We develop a plugin for Pin so that we can log the executed instructions (i.e., traces). Traces usually begin at a taken branch and end with an unconditional branch, including calls and returns. We utilize another Pin plugin to perform the taint analysis on the trace. As previously mentioned, only instructions that are relevant to the inputs are kept after the taint analysis. Furthermore, we employ the commonly used binary analysis tool BAP ([25]) to lift assembly instructions of the trace into an intermediate representation (IR) specified by BAP [26]. BAP only supports a subset of x86, which does not include floating point and privileged instructions. This is one of the limitations of the current implementation.

Trace-based symbolic execution interprets instructions along the trace one after another, and generally speaking, both forward and backward symbolic execution can capture path constraints during this step. Our preliminary implementation shows that symbolic execution based on the Dijkstra and Flanagan-Saxe style weakest precondition generally yields more succinct formulas in our context [16], hence, for our implementation, we adopt the weakest precondition technique [27] to build our symbolic formula.

To implement the symbolic execution, we employ the trace analysis component of BAP, which provides weakest precondition computation. Furthermore, we use a common SMT solver (STP [28]) to solve constraints generated during path deviation detection (§III-D) and path correspondence analysis (§III-E).

Guarded Command Language. Weakest preconditions are calculated over programs written in the Guarded Command Language (GCL). Before we calculate the weakest precondition, BAP translates the IR into this GCL as specified below:

$$\begin{array}{rcl} S,\, T & ::= & Id := Expr \\ & | & \texttt{assert}\ Expr \\ & | & \texttt{assume}\ Expr \\ & | & S\ ;\ T \\ & | & S \,\square\, T \end{array}$$

The language contains assignment statements: x:=E, which sets the value E to program variable x. The `assert` and `assume` statements have no effect if the given expression is evaluated as `true`; otherwise, the `assert` statement fails and the `assume` statement blocks the execution. The operational semantic is defined such that every step either blocks, fails, or terminates. The statement S ; T is the sequential composition of S and T where T executes only after S terminates. $S \square T$ is the nondeterministic choice between S and T, and it is equivalent to `(assume B;S) (assume ¬B;T)`. Although the GCL looks simple, it is expressive enough to convert x86 assembly into this language [29]. The translation is straightforward and is discussed elsewhere [30].

Weakest Precondition. The weakest precondition method builds the constraints of inputs to characterize programs [31] and execution traces [16]. Let Q be the postcondition of a program, while the weakest precondition $\mathrm{wp}(P, Q)$ of a program P in relation to the postcondition Q is a boolean formula F over the input space of the program such that if F holds, then the program execution must satisfy Q.

We denote the weakest precondition of a program trace as $\mathrm{wp}(T, Q)$, where T is the execution trace and the postcondition Q is the program output. Assume trace T contains a sequence of instructions $< i_1, i_2, ..., i_n >$; the calculation of $\mathrm{wp}(T, Q)$ can thus be inductively defined. That is, we first calculate $\mathrm{wp}(i_n, Q) = Q_{n-1}$, and then $\mathrm{wp}(i_{n-1}, Q_{n-1}) = Q_{n-2}$, until $\mathrm{wp}(i_1, Q_1) = Q_0$.

Processing Loops. As in previous works [32], [33], we calculate the weakest precondition targeting acyclic GCL programs converted from the IR. If program invariants are available, the loops will be converted directly into GCL [30]; otherwise, we consider bounding the loop iterations to make the program acyclic. Then we can calculate the predicates automatically based on the program syntax. In BAP's implementation, the `-unroll` option unrolls the loop to a maximum of 32 iterations while checking the loop conditions and eliminating the back edge. For example, for the loop `while` e S, by expanding one iteration, we get `(assume` e`;` S`;` `while` e S`)` $\square$ `assume` $\neg e$. We can expand the program as many times as we would like and bound the loop iteration with an `assert` statement.

Availability. We have released the tool[1] for public dissemination.

V. Discussion

We have proposed a program-semantics-based method for autograding programming assignments. Our method relies on detecting semantic differences between the reference implementation and student submissions. While program equivalence is generally undecidable, we have developed a practical tool with the proposed method. Our method is superior to the more common test-based methods in terms of the reduced effort to create test suites. It also tackles the soundness issue of test-based methods and detects false negatives of online platform judges. Moreover, it outperforms other reference-based methods that require an additional error model to be specified. Another advantage is that our method is resilient to program variations. Theoretically, it can handle most cases even if student submissions often vary. In this section, we analyze several typical variations and explain how our method overcomes these challenges.

As previously mentioned, weakest precondition calculation is the core technique that we leverage to perform symbolic execution. Before discussing the technique's resilience to program variations, we introduce the three main properties of the weakest precondition: *Commutative*, *Combinative*, and *Associative*, as presented in Table II.

A. Renaming

Students may use different names for the same variable. However, since we compile source code into program binaries and directly analyze the program execution traces, all the names are pruned in the assembly instructions in the program binaries (and further in the execution traces). That is, our method is resilient to renaming.

B. Noise Instruction

Noise instructions are common in submissions of students in the type of entry-level programming courses that we target. To solve a problem, students usually try every method they can think of and revise over and over again. As a result, redundant instructions remain. Suppose an irrelevant instruction I_1 is inserted after instruction I_0. Our tool computes the weakest precondition $\mathrm{wp}(I_0; I_1, Q)$. Because I_1 is an irrelevant statement, we have $\mathrm{wp}(I_0; I_1, Q) = \mathrm{wp}(I_0, Q)$. This equation can be extended to the case of multiple irrelevant instructions, $\mathrm{wp}(I_0; I_1; ...I_n, Q) = \mathrm{wp}(I_0, Q)$, where I_1 to I_n are irrelevant instructions. Therefore, our method is resilient to noise instructions.

C. Statement Reordering

Statement reordering specifies the case when the order of two or more instructions without control or data flow dependence is rearranged. Suppose we consider two such instructions, I_0 and I_1. The computed weakest precondition is $\mathrm{wp}(I_0; I_1, Q)$. But when we switch these two instructions, the computed weakest precondition becomes $\mathrm{wp}(I_1; I_0, Q)$. We have $\mathrm{wp}(I_0; I_1, Q) = wp(I_1; I_0, Q)$ according to the *Commutative property*, when the two instructions are independent. In addition, this equation still holds when multiple instructions are reordered; in fact, such a problem can be divided into several subproblems, each of which is a reordering of two instructions. Therefore, our method is resilient to statement reordering.

[1]https://github.com/s3team/AutoGrader

TABLE II
PROPERTY FOR WEAKEST PRECONDITION CALCULATION

Commutative :	$\mathrm{wp}(I_0; I_1, Q) = \mathrm{wp}(I_1; I_0, Q)$ if I_0, I_1 are independent
Combinative :	$\mathrm{wp}(I_0, Q) = \mathrm{wp}(I_1; I_2, Q)$ if I_0 is semantically equivalent to $I_1; I_2$
Associative :	$\mathrm{wp}(I_2, \mathrm{wp}(I_1; I_0, Q)) = \mathrm{wp}(I_2; I_1, \mathrm{wp}(I_0, Q))$

D. Loop Unwinding

Complete or partial loop unwinding is common among student submissions. We often observe that students tend to compute boundary conditions separately from a loop; therefore, their submissions vary from each other. However, our method is based on dynamic execution traces where the loops are all completely unwound. As a result, our method is resilient to loop unwinding.

E. Instruction Splitting

Instruction splitting refers to the case where an instruction I_0 can be split into two instructions I_1 and I_2 in student submissions. This is commonly found in students' assignments. The computed weakest precondition for the two instructions is $\mathrm{wp}(I_1; I_2, Q)$, which is equivalent to $\mathrm{wp}(I_0, Q)$ according to the *Combinative property*, as long as I_1, I_2 are semantically equivalent to I_0. This also applies to cases in which one instruction is split into multiple instructions. Moreover, instruction aggregation can be viewed as the reverse of instruction splitting, which can also be addressed by our technique. Therefore, our method is resilient to instruction splitting and aggregation.

F. Trace Splitting

Trace splitting refers to equivalent paths in traces. Two paths are equivalent if they yield the same computational result. Normally we see such cases when students use different calculation processes for some boundary conditions. Despite the difference in executions, they achieve the same computational results as the reference implementations. For example, when students attempt to compute the absolute value of x, they either check the condition $x >= 0$ or the condition $x > 0$. In this case, both solutions are correct, but the boundary case $x = 0$ can cause a path deviation. Conceptually similar to what we have described in Section III-E, trace splitting can cause "deviations" that lead to false positives in grading. Our preliminary studies show that this is actually a common case in student submissions. Thus, we propose a *trace equivalence checking* technique (§III-E) to detect cases where path deviations actually result from trace splitting. With this technique, our method is resilient to trace splitting.

VI. EVALUATION

We have implemented our method using a tool called AUTOGRADER. The tool consists of three main stages: (i) input generation, (ii) deviation detection, and (iii) equivalence checking, as shown in Figure 2. The purpose of AUTOGRADER is to provide feedback: *correct* or *incorrect with counterexamples* for programming assignments. In this section, we present an evaluation of AUTOGRADER.

A. Benchmarks

We collected the submitted programs for 10 programming problems from *codechef.com* [10]; the programming programs are listed in Table III. *codechef.com* is an online platform for beginners to practice programming. It will provide immediate feedback using a test case-based methodology. From *codechef.com*, we selected beginner-level problems for which there existed clear specifications to construct correct reference implementations. Before we fed a submission to AUTOGRADER, we customized it to take test arguments from the standard input, so that we can apply our tracer to it. In addition, we only selected those submissions that had same input types. This is common in student assignments where function inputs and outputs are specified by instructors. We also eliminated a small number of submissions that contained type cast and multi-thread that may affect our analysis.

Note that each collected submission was associated with the ground truth of its correctness. If the submission was incorrect, we collected a detailed error message to explain the reason for its failure. Typical reasons were that compilation error, runtime error, and incorrect outputs. Considering our research context, we only selected submissions that were compilable and executable. Hence, incorrect programs were selected because of logic errors (i.e., incorrect outputs). The total number of submissions was 10,270 for 10 different problems, among which 3,351 were correct submissions and 6,919 were incorrect ones. The average number of lines of code each of these attempts, which ranged from 27 (*CHEFKEY*) to 73 (*BOOKCHEF*), was 46.

B. Evaluation Metric

A detailed breakdown of the number of examples for each benchmark problem is reported in Table III. We evaluated our tool with respect to *effectiveness* and *performance*. *Effectiveness* measures the fraction of correct judgments that are made by our tool compared with the ground truth. *Performance* is reported to show the cost of our proposed technique.

C. Effectiveness

As aforementioned, considering the potential path explosion problem of our white-box fuzzer, we set up a time limit (five minutes) to grade a submission. It is reasonable that in practice, people will not wait that long to receive a feedback report for a program submission. This five-minute threshold would include the whole analysis process; that is, multiple rounds of input generation, deviation detection, and equivalence checking.

Table III reports the effectiveness of our method. *Accuracy* is defined as the portion of correctly classified cases. The overall accuracy of our system was *92.80%*, and we found

TABLE III
BENCHMARK PROBLEMS

Problem	Statistics				Effectiveness		Performance				
	# Correct	# Incorrect	LoC	# Timeout	Accuracy	E-Accuracy	IG (s)	PD (s)	EC (s)	# Iter.	Total (s)
1. ALEXTASK	550	546	51	172	83.12%	98.81%	0.87	3.86	1.89	4	25.48
2. CHEFAPAR	262	454	29	14	95.53%	97.48%	0.62	2.27	1.81	3	13.82
3. ENTEXAM	203	1719	61	116	93.65%	99.68%	1.03	2.31	1.20	5	32.99
4. NOTINCOM	675	351	34	145	85.76%	99.90%	0.60	3.50	1.91	4	16.61
5. ANKTRAIN	419	387	41	22	96.52%	99.25%	0.67	3.04	3.27	2	14.02
6. TRISQ	445	367	27	14	96.06%	97.78%	0.47	2.26	1.81	3	12.24
7. KOL16B	262	437	35	29	94.56%	98.71%	0.59	1.81	1.82	3	13.46
8. RGAME	238	723	41	50	93.86%	99.06%	0.71	1.69	1.78	3	15.89
9. BOOKCHEF	77	208	73	40	85.61%	99.65%	1.23	3.86	1.50	6	30.71
10. CHRL4	220	1727	49	137	89.86%	97.49%	0.92	2.18	1.92	3	24.34
Total	3351	6919	46	794	92.80%	98.62%	-	-	-	-	-

(IG: Input Generation, PD: Path Deviation, EC: Equivalence Checking.)

that most of the tests were indeed cut off due to the time limitation. Further study of the properly graded submissions showed promising results; we found that there was **no** false negative found in our evaluation. In other words, no incorrect submission is graded as correct within a reasonable time limit, which is critical for a practical tool.

By further studying the evaluation results, one interesting observation is that all the time-out submissions were labeled *correct* according to the ground truth. Although our method does not theoretically cover paths of the whole program, in practice, it is reasonable to consider a submission correct if AUTOGRADER is unable to yield path deviations within a considerable amount of time. Given this observation, we refined the decision procedure of AUTOGRADER to mark a submission as correct whenever its analysis reached the time limit. We further re-graded the time-out benchmarks as "correct". The measurements are reported as *Extended Accuracy (E-Accuracy)* in Table III. Overall, our tool achieved an accuracy of *98.62%*.

In what follows, we discuss two reasons that for the potential mis-classifications of correct/incorrect submissions: *Incomplete Test Suites* and *Same Output Coincidence*.

1) Incomplete test suite: An incomplete test suite is a typical problem that may cause our method to fail. While browsing the forums of online grading platforms, we noticed the issue of "not having comprehensive test suites" was very common [34]–[36].

The incompleteness of test suites is primarily because of two reasons, theoretical limit and practical workload. Theoretically, it is impossible to generate test inputs that can fully cover all possible behaviors of a program. In addition, to reduce the workload of the online grading services, a limited number of test cases that deliver a reasonable processing time is usually adopted.

Interestingly, we found 11 false positives of AUTOGRADER. Further study showed that these false positives are in fact false negatives of the testing-based approach used in *codechef.com*. That is, buggy submissions were labeled as "correct" in the ground truth because of incomplete test cases.[2]

[2]Our system detected 11 false negatives. The submission numbers are: 12318552, 12320180, 12271123, 12302502, 12312102, 12330737, 12330861, 12330880, 12330906, 12640689, and 9912397.

```
1 max3(int a, int b, int c) {
2   if (a>=b && a>=c)
3     return a-b;
4   else if (b>=a && b>=c)
5     return b;
6   else
7     return c;
8 }
```

2) Same output coincidence: As previously discussed, AUTOGRADER reports incorrectness when: 1) different outputs are triggered, or 2) the same output but true path deviations are detected. True path deviations are guaranteed by the conjunction of different output formulas and different path formulas. However, there is still a small chance that the same output value is yielded and no path deviation is detected given an incorrect submission. For example, consider the listing above: The bug of this submission is on *line 3*, where the correct return value should be a but the buggy implementation returns $a - b$. If AUTOGRADER uses inputs $a = 3, b = 0, c = 1$, the corresponding output of the incorrect program will be the same as the reference. Therefore, AUTOGRADER will mis-grade this incorrect submission as correct. Nevertheless, as reported in our evaluation of real-world programming submissions, **no** false negative was found, which indicates that the "same result coincidence" problem is not likely in practice.

D. Performance

We also evaluated the performance of the proposed framework. We conducted the evaluations on a desktop with a Xeon E5-1607 3.00GHz CPU and 4GB memory running 64-bit Ubuntu 12.04 LTS. We set up the evaluation and recorded the average time for one iteration, as well as the number of iterations for which the tool detected deviations. We report our experimental results in Table III and Figure 5.

For the 10 problem sets we analyzed, the average time for one iteration for each problem, including *input generation*, *path deviation*, and *equivalence checking*, ranged from 4.19 seconds to 6.99 seconds. The average iteration number for each problem to detect deviations ranged from 2 to 6. Of the incorrect submissions, 73.65% were detected erroneous within 5 iterations and 99.78% were within 20 iterations. This result indicates a fast convergence over the analyzed data. For 5-iteration detection, the problem *ANKTRAIN* had the highest percentage (97.16%) of incorrect submissions detected. The solution to this problem was a case-by-case analysis that contained 9 conditional branches. Student submissions were decided incorrect mostly within the 9 branches; only a small

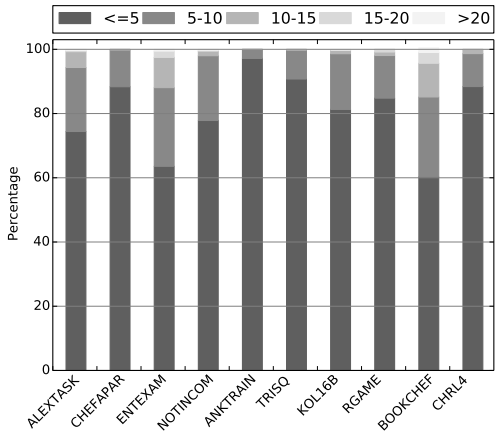

Fig. 5. Performance for AUTOGRADER

portion of them were over 10 iterations due to equivalence checking. The problem *BOOKCHEF* had the lowest percentage (64.92%) of submissions that were detected as erroneous within 5 iterations. The solution to this problem had nested loops which cause a longer analysis time.

At this point, we have only tested our framework on an entry-level dataset for which the generated traces were short to analyze and the formulas were easier to solve. Additionally, despite the soundness of one trace, our method is incomplete in terms of the whole program. It is future work to improve the scalability and completeness of the tool so that it is more practical to tackle more complicated problems.

VII. RELATED WORK

We have proposed a formal-semantics-based method to determine the correctness of programming problem submissions. Our research seeks to identify semantic-level differences in input programs that can lead to potential path deviations. In this section, we review related work on autograding and discuss existing methods for detecting semantic differences in programs.

A. Automatic Grading

The automatic assessment of programming assignments has been studied over years, with many tools being presented. The majority of such systems are testing-based [3]. In general, researchers have mainly focused on checking functional correctness or, in other words, the program validity [37]. Targeted platforms include Java [6], [9], web applications [8], and domain-specific languages [4]. Often automatic assessment is done by examining program behaviors on a set of test inputs [38]–[40]. The test cases can be either manually crafted or automatically generated [4], [5]. With the proposed systems, inputs that fail the tests are provided as feedback. However, generating a set of test inputs that can cover all the possible errors in student submissions is impossible. Recent studies have made contributions to test oracle improvement based on search-based test generation [41]–[44]. Yet most of today's automated tools rely on the instructor to provide sets of test inputs.

While black-box testing tests a program with input-output pairs, white-box testing tests the internal structure of a program. Some studies have proposed using symbolic execution [45]–[47] to automatically generate high-coverage test suites. Symbolic execution has been applied to automated grading. With symbolic execution engines such as KLEE [32], researchers reuse industrial-grade automatic testing tools for automated grading to provide timely feedback on students' C [48] and JavaScript [5] program submissions.

Another popular approach to automatic grading measures the similarity between abstract representations of a student's submission and corresponding reference implementations [49]–[51]. Although promising, the theoretical elegance of this approach is overtaken by the uncertainty. The detection of a similarity between a program and reference implementations is unable to provide students with hints for the root causes that lead to the failure. Additionally, variations in reference implementations are challenging to address.

Researchers have also proposed some machine learning-based autograding methods. These methods rely on features extracted from the basic descriptions of a program, (e.g., its control flow). By adopting clustering [52]–[54] and classification [55] algorithms, these systems can determine the correctness of a submission. However, our study shows that the overall performance of such machine learning-based techniques is not as effective as our method.

B. Program Semantic Difference Detection

We also reviewed a number of techniques that identify semantic differences among programs based on either static or dynamic analysis. Many of these techniques are designed for tracking software evolution. Jackson and Ladd [56] capture dependencies between inputs and outputs, but their work does not leverage any theorem-proving technique. More recent analyses have been based on regression verification [57], which uses SMT solvers to check the equivalence of C programs. Some progress has been made by SYMDIFF [58], which converts source code to an intermediate verification language and then identifies semantic differences. In contrast to previous work, this tool expresses the results in terms of the observable input-output behavior of programs, rather than syntactic structures.

Symbolic execution has been widely used in recent research to capture execution semantics. Differential symbolic execution [59] detects semantic differences by using symbolic execution to enumerate paths and check equivalence. UCKLEE [60] synthesizes inputs to two C programs and uses bit-accurate symbolic execution to verify that the produced formulas. These two methods are conceptually similar to AUTOGRADER, but they focus on source code. We also observe similar ideas in studies of fine-grained analysis of binaries. BinHunt [61] and iBinHunt [62] perform symbolic execution within basic blocks and verify the generated formulas represent input-output relations. However, these tools are insufficient to identify similarities or differences across basic blocks. Therefore, trace-oriented analyses are proposed

to identify semantic differences beyond the ability of basic blocks [16], [17].

VIII. Limitations and Future Work

In this section, we discuss the limitations of AutoGrader and suggest directions for future work.

First, AutoGrader only determines the correctness of programming assignments and provides counterexamples. In other words, our method decides the functional correctness. Off-the-shelf automatic graders usually rely on the number of passed tests to mark a submission's "grade" in addition to its correctness. However, a meaningful grading system should identify the shortest distance between the correct implementation and submissions. Traditional grading methods do not represent the score in this way and rely heavily on the quality of test suites. On the contrary, our method conducts the analysis with finer granularity. In the future, we plan to leverage model counters [63], which approximate the number of satisfiable models for boolean formulas, or similar techniques to calculate the distance between submissions and the correct reference. Moreover, on top of the recorded trace, we plan to detect the redundant code in a program and use this information to grade assignments in terms of "elegance".

Second, AutoGrader is limited by the capability of constraint solver and symbolic execution tools. During the path deviation detection process, if the constraint solver finds a *sat* assignment to the formula, it is truly satisfiable. However, an output of *no* may mean the formula is *unsat*, or the solver cannot find a satisfiable assignment limited by its capability. This can potentially lead to false negatives. We tackle the problem by iterating the process for many rounds, thereby reducing the probability of such cases. A similar situation may occur during the equivalence checking process, and our tool would theoretically report false positives. In our experiments, we have not seen such false positives or false negatives. Finally, AutoGrader suffers from the common limitations of current symbolic execution tools, such as that they cannot perform non-linear arithmetic operation or floating point calculation.

IX. Conclusion

We have proposed a formal-semantics-based approach for automatically grading programming assignments with a *single* correct reference implementation provided by the instructor. By searching for inputs that can lead to path deviations between a student's submission and the reference implementation, we can determine the correctness of the submission and provide counterexamples for an incorrect submission. We have implemented this novel approach using a tool called AutoGrader and evaluated our technique over a dataset containing 10,270 submissions collected from an online programming site. The experiment revealed that our proposed method yielded no false negatives, while the online programming site yielded 11 false negatives due to incomplete test suites.

X. Acknowledgment

We thank the reviewers for their thorough comments and suggestions. The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

References

[1] S. Carson, "MOOC," https://ocw.mit.edu/about/media-coverage/press-releases/chi600intro-announcement/, 2013, [Online; accessed 21-March-2017].

[2] D. S. Weld, E. Adar, L. Chilton, R. Hoffmann, E. Horvitz, M. Koch, J. Landay, C. H. Lin, and Mausam, "Personalized online education - a crowdsourcing challenge," in *AAAI Workshop - Technical Report*, vol. WS-12-08. AAAI, 2012, pp. 159–163.

[3] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 2010, pp. 86–93.

[4] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 1117–1126.

[5] L. Gong, "Auto-grading dynamic programming language assignments," University of California, Berkeley, Tech. Rep., 2014.

[6] M. T. Helmick, "Interface-based programming assignments and automatic grading of Java programs," in *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '07. ACM, 2007, pp. 63–67.

[7] M. Sztipanovits, K. Qian, and X. Fu, "The automated web application testing (AWAT) system," in *Proceedings of the 46th Annual Southeast Regional Conference*, ser. ACM-SE 46. ACM, 2008, pp. 88–93.

[8] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu, "APOGEE: Automated project grading and instant feedback system for web based computing," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08, 2008, pp. 77–81.

[9] M. Joy, N. Griffiths, and R. Boyatt, "The BOSS online submission and assessment system," *J. Educ. Resour. Comput.*, vol. 5, no. 3, Sep. 2005.

[10] CodeChef, "CodeChef," https://www.codechef.com/, 2009, [Online; accessed 21-April-2017].

[11] CodeForces, "CodeForces," https://hackerrank.com/, 2009, [Online; accessed 21-April-2017].

[12] Hackerrank, "Hackerrank," https://www.hackerrank.com/, 2017, [Online; accessed 21-April-2017].

[13] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, 2013, pp. 15–26.

[14] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 41–51.

[15] S. Gulwani, I. Radicek, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *CoRR*, vol. abs/1603.03165, 2016. [Online]. Available: http://arxiv.org/abs/1603.03165

[16] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. USENIX Association, 2007, pp. 15:1–15:16.

[17] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, Nov 2014, pp. 66–77.

[18] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.

[19] A. Sharma, "Pathgrind," https://github.com/codelion/pathgrind, 2013, [Online; accessed 21-March-2017].

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, no. 6. ACM, 2005, pp. 213–223.

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2008, pp. 151–166.

[22] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 190–200.

[25] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011, pp. 463–469.

[26] ——, "BIL," https://github.com/BinaryAnalysisPlatform/bil, 2017.

[27] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1997.

[28] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 519–531.

[29] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*. IEEE, 2007, pp. 311–325.

[30] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: Generating compact verification conditions," in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '01. ACM, 2001, pp. 193–205.

[31] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[32] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

[33] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 206–215.

[34] leetcode, "Not enough test cases," https://discuss.leetcode.com/topic/22381/not-enough-test-cases/2?page=1, 2015, [Online; accessed 21-April-2017].

[35] ——, "Majority Element - Not enough test cases," https://discuss.leetcode.com/topic/77846/majority-element-not-enough-test-cases, 2017, [Online; accessed 21-April-2017].

[36] ——, "Bug report, not enough test cases," https://discuss.leetcode.com/topic/58802/bug-report-not-enough-test-cases, 2017, [Online; accessed 21-April-2017].

[37] J. B. Hext and J. Winings, "An automatic grading scheme for simple programming exercises," *Comm. ACM*, vol. 12, no. 5, pp. 272–275, 1969.

[38] M. Wick, D. Stevenson, and P. Wagner, "Using testing and JUnit across the curriculum," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '05. ACM, 2005, pp. 236–240.

[39] u. von Matt, "Kassandra: The automatic grading system," *SIGCUE Outlook*, vol. 22, no. 1, pp. 26–40, Jan. 1994.

[40] M. Joy, N. Griffiths, and R. Boyatt, "The BOSS online submission and assessment system," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, p. 2, 2005.

[41] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 247–258.

[42] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[43] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.

[44] F. G. de Freitas and J. T. de Souza, "Ten years of search based software engineering: A bibliometric analysis," in *Proceedings of the Third International Symposium on Search Based Software Engineering*, 2011, pp. 18–32.

[45] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[46] G. Birch, B. Fischer, and M. Poppleton, "Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs," *Software & Systems Modeling*, vol. 1, 2017.

[47] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 4, pp. 339–353, 2009.

[48] J. Gao, B. Pang, and S. S. Lumetta, "Automated feedback framework for introductory programming courses," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2016, pp. 53–58.

[49] G. Michaelson, "Automatic analysis of functional program style," in *Australian Software Engineering Conference, 1996., Proceedings of 1996*. IEEE, 1996, pp. 38–46.

[50] K. Ala-Mutka, T. Uimonen, and H.-M. Jarvinen, "Supporting students in C++ programming courses with automatic program style assessment," *Journal of Information Technology Education*, vol. 3, no. 1, pp. 245–262, 2004.

[51] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments," *Information and Software Technology*, vol. 55, no. 6, pp. 1004–1016, 2013.

[52] D. Perelman, J. Bishop, S. Gulwani, and D. Grossman, "Automated feedback and recognition through data mining in code hunt," Microsoft, Tech. Rep., 2015.

[53] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, "Automatic grading and feedback using program repair for introductory programming courses," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. ACM, 2017, pp. 92–97.

[54] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016, pp. 39–40.

[55] S. Srikant and V. Aggarwal, "Automatic grading of computer programs: A machine learning approach," in *Proceedings of the 12th International Conference on Machine Learning and Applications*, vol. 1. IEEE, 2013, pp. 85–92.

[56] D. Jackson, D. A. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications." in *ICSM*, vol. 94. ACM, 1994, pp. 243–252.

[57] B. Godlin and O. Strichman, "Regression verification," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. ACM, 2009, pp. 466–471.

[58] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 712–717.

[59] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 226–237.

[60] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 669–685.

[61] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[62] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[63] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.