Measuring Code Behavioral Similarity for Programming and Software Engineering Education

Sihan Li[1] Xusheng Xiao[2] Blake Bassett[1] Tao Xie[1] Nikolai Tillmann[3]
[1]University of Illinois at Urbana-Champaign, Champaign, IL, USA
[2]NEC Laboratories America, Princeton, NJ, USA
[3]Microsoft Research, One Microsoft Way, Redmond, WA, USA
[1]sihanli2,rbasset2,taoxie@illinois.edu, [2]xsxiao@nec-labs.com, [3]nikolait@microsoft.com

ABSTRACT

In recent years, online programming and software engineering education via information technology has gained a lot of popularity. Typically, popular courses often have hundreds or thousands of students but only a few course staff members. Tool automation is needed to maintain the quality of education. In this paper, we envision that the capability of quantifying behavioral similarity between programs is helpful for teaching and learning programming and software engineering, and propose three metrics that approximate the computation of behavioral similarity. Specifically, we leverage random testing and dynamic symbolic execution (DSE) to generate test inputs, and run programs on these test inputs to compute metric values of the behavioral similarity. We evaluate our metrics on three real-world data sets from the PEX4FUN platform (which so far has accumulated more than 1.7 million game-play interactions). The results show that our metrics provide highly accurate approximation to the behavioral similarity. We also demonstrate a number of practical applications of our metrics including hint generation, progress indication, and automatic grading.

1. INTRODUCTION

In recent years, with the advance of information technology, online programming and software engineering education has gained a lot of popularity. Introductory programming courses are among the most popular Massive Open Online Courses (MOOC). Many MOOC providers, such as edX, Coursera, and Udacity, have offered programming and software engineering courses, in which thousands of students worldwide are enrolled. In addition to MOOC, other forms of online programming and software engineering education have also attracted a lot of attentions. For instance, PEX4FUN [28], along with its recent successor Code Hunt [6], are online educational platforms for teaching and learning programming and software engineering via gaming. So far the PEX4FUN platform has had over 1.7 million game-play interactions made by players around the world.

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889204

One challenge of online programming and software engineering education is to allow large-scale classes while maintaining the quality of the education. Typically, popular courses often have hundreds or even thousands of students but only a few course staff members. The limited number of staff members poses challenges in providing quality education. On instructors' side, many tasks, such as grading and providing customized feedback on programming assignments, require instructors to go through and understand students' code. The workload of these tasks is prohibitively huge. However, skipping or delaying such tasks prevents instructors from keeping track of students' performance. On students' side, it is difficult for them to get prompt, customized feedback and help from the instructors. Although students may seek help from peers, peers are often not capable of helping or providing valuable feedback in many cases. Instructors or peers cannot always sit with students while the students are coding or provide prompt hints when the students encounter problems.

To alleviate this issue, we propose using tool automation to reduce instructors' burden in the teaching process and improve students' learning experiences in programming and software engineering courses. We envision that the capability of automatically quantifying the *behavioral similarity*[1] between programs can be helpful in many aspects. For instance, it can assist instructors in evaluating students' programs. The behavioral similarity between students' programs and a sample solution program can be used as a factor in grading. Higher similarity generally implies a higher score. In addition, it can be used as a progress indicator during students' coding process. Our tool automation can constantly compute and display the behavioral similarity between the students' current program and the solution program so that students can keep track of their progress. A series of increasing similarity scores achieved by a student indicates that the student is moving towards the solution while a significant drop may indicate that the student is modifying the code in a wrong way or reverting correct code. Such information can help students realize potential problems early and avoid further deviations. Moreover, behavioral similarity can also be used to detect similar programs across different students. These similar programs from other students can be used to generate hints for the next step when students need help on some task. We describe such broad applications in detail in Section 6.

[1]Behavioral similarity is also referred to as semantic similarity. In contrast to syntactical similarity, behavioral similarity concerns about input/output behaviors of programs.

However, in general, it is challenging to precisely quantify the behavioral similarity. Ideally, the behavioral similarity between two programs can be measured by computing the proportion of inputs producing the same output on both programs (referred to as agreed inputs) over the entire input domain. A straightforward approach that enumerates all inputs in the input domain and runs each input against both programs to compare the outputs is impractical or infeasible for programs with a large or infinite input domain. To practically measure behavior similarity, we propose running representative inputs instead of all inputs on programs to approximate their behavioral similarity based on two insights. The first insight is that by uniformly sampling a significant portion of inputs from the input domain, the behavioral similarity computed based on the sampled inputs can provide a good approximation to the actual behavioral similarity. The second insight is that using inputs that exercise different program paths as representatives can also provide a good approximation, as these different program paths usually represent different program behaviors. Dynamic symbolic execution (DSE) [11,23,26] is a technique that executes a program both symbolically and concretely to collect constraints from branches and systematically negates part of the collected constraints to generate new program inputs. With the advent of powerful constraint-solving tools [9], DSE has shown promising results in generating test inputs that achieve high code coverage. We can leverage DSE to systematically generate high-covering test inputs, and use the generated test inputs to explore program paths efficiently.

Based on these two insights, we present three metrics for using test generation techniques to measure behavioral similarity. Our first metric, *Random Sampling* (RS), is computed by using random test generation to generate inputs uniformly distributed over the input domain. Then we run both programs separately on each test input to compare their outputs. The proportion of the agreed inputs over the sampled inputs is the value for RS. The metric *Single-program Symbolic Execution* (SSE), based on the second insight, is computed by using one program as the *reference program*, and employing DSE to generate test inputs that capture the behaviors of the reference program. We then run the other program under analysis on these test inputs and compute the proportion of agreed inputs over the generated inputs as the value of SSE. Since the test generation is based on only the reference program and not sensitive to the program under analysis, the generated test inputs may fail to exercise some behaviors in the program under analysis. Our third metric, *Paired-program Symbolic Execution* (PSE), addresses this limitation by constructing a paired program [25] from the two programs. The input domain of the paired program is exactly the same as that of the two programs. The paired program runs the two programs on the same input and asserts their outputs to be the same. Then we leverage DSE to generate test inputs on the paired program and compute the proportion of test inputs that pass the assertion over the generated inputs as the PSE value. Since the test generation for PSE is performed on the paired program, which is constructed from both programs, PSE is also sensitive to the program under analysis and likely to reveal more behavioral differences than SSE does.

We implement a tool to measure these three metrics of behavioral similarity for C# programs. We implement random test generation to compute RS, and use Pex [26,27] (a state-of-the-art DSE engine recently released as the IntelliTest feature in Visual Studio 2015) to implement SSE and PSE. We evaluate our metrics on student code artifacts produced in three real-world courses from the Pex4Fun platform. The evaluation results show that, in general, RS provides highly accurate approximations to the behavioral similarity, but it may not be able to distinguish small behavioral differences. As a complement, PSE is highly effective in detecting such small behavioral differences but does not provide as good approximations as RS does. Such results suggest that these two metrics can be used in a complementary way to provide better approximation. We also demonstrate practical applications of our metrics including hint generation, progress indication, and automatic grading.

Our paper makes the following major contributions:

- Three metrics that accurately quantify the behavioral similarity between programs. These metrics can complement each other in producing further better results.
- An open source tool that implements these three metrics based on the Pex tool. Our tool can be accessed from the project website[2].
- Evaluations on code artifacts from three real-world courses on the Pex4Fun platform, showing that our metrics RS and PSE are highly effective in measuring behavioral similarity.
- Demonstrations of applying our three metrics for programming and software engineering education.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the test generation techniques used in our metrics. Section 3 gives an overview of measuring behavioral similarity, including some definitions and illustrating examples. Section 4 describes our metrics in detail. Section 5 presents our evaluation results. Section 6 demonstrates practical applications of our metrics to programming and software engineering education. Section 7 discusses major threats to validity. Section 8 discusses related work and Section 9 concludes the paper.

2. BACKGROUND

Random Test Generation. Typically, random test generation treats the program under test as a black box and randomly chooses inputs from the input domain of the program. A straightforward technique of random test generation is to generate inputs by randomly sampling a value from the input domain according to the uniform distribution. There are also other advanced strategies for sampling inputs, e.g., adaptive random testing [7] and feedback-based techniques [21] for object-oriented programs. Since random test-generation techniques do not analyze the program, their costs are relatively low, making random test generation efficient. However, a common limitation of random techniques is that they often fail to generate inputs for corner cases. For our random metric, we use a simple random technique to uniformly sample inputs from the input domain.

DSE-based Test Generation. DSE [11,23,26] explores feasible paths of the program under test and generates test inputs to exercise the paths. DSE starts with an arbitrary or default concrete input, and symbolically executes the program along the path led by the concrete input. During the execution over the path, DSE collects the conditions from

[2]https://sites.google.com/site/behavsim/

```
public static string isFancyYear(int i) {
    if (i < 1000 || i > 9999)
        return "not a fancy year";
    int digit = i % 10;
    while (i != 0) {
        if (i % 10 != digit)
            return "not a fancy year";
        i /= 10;
    }
    return "fancy year";
}
```

(a) A program that checks "fancy year" using a loop

```
public static string isFancyYear(int i) {
    if (i == 1111) return "fancy year";
    if (i == 2222) return "fancy year";
    if (i == 3333) return "fancy year";
    if (i == 4444) return "fancy year";
    if (i == 5555) return "fancy year";
    if (i == 6666) return "fancy year";
    if (i == 7777) return "fancy year";
    if (i == 8888) return "fancy year";
    if (i == 9999) return "fancy year";
        return "not a fancy year";
}
```

(b) A program that checks "fancy year" by enumerating valid values

Figure 1: An example of behavioral equivalence

the taken branches as the path constraint. Then DSE systematically negates part of the path constraint to form new path constraints, and leverages constraint solvers to solve these path constraints for obtaining new test inputs. These new test inputs steer the future explorations towards different paths of the program. Iteratively, DSE collects new path constraints and generates new test inputs for achieving high structural coverage, such as statement coverage and branch coverage. With the advances of research on constraint solvers, e.g., Z3 [9], DSE-based tools, such as Pex [26] and SAGE [12], have shown promising results in structural testing and security/system testing [12, 23, 26].

3. OVERVIEW

To quantify the behavioral similarity of two programs, we formally define the behavioral similarity and related concepts, and use examples to illustrate these definitions. We start with formally defining program execution.

Definition 1 (Program Execution). *An execution of a program P is a function $exec : P \times I \rightarrow O$ that maps an input $i \in I$ to an output $o \in O$, where I is the input domain of P and O is the output domain of P.*

Using the definition of program execution, we further define behavioral equivalence and difference between programs as well as behavioral similarity that measures how similar the behaviors of two programs are on the input domain.

Definition 2 (Behavioral Equivalence). *Two programs P_1 and P_2 that share the same input domain I are behaviorally equivalent, denoted as $exec(P_1, I) = exec(P_2, I)$, iff $\forall i \in I, exec(P_1, i) = exec(P_2, i)$.*

Figure 1 shows two programs that implement the same functionality, i.e., behave equivalently. These two programs accept an integer as input, and return a string "fancy year" if the number is between 1000 and 9999, and all digits of the number are the same. The first program, in Figure 1a, implements this functionality via explicit bound checking and a loop iterating through each digit. The second program, in Figure 1b, enumerates all possible fancy years and checks the input against these values. A cursory inspection of the two programs can determine that on all possible inputs, the outputs of the two programs always agree. Although the two programs syntactically look very different, they *behave* equivalently.

Definition 3 (Behavioral Difference). *Two programs P_1 and P_2 that share the same input domain I are behaviorally different, denoted as $exec(P_1, I) \neq exec(P_2, I)$, iff $\exists i \in I, exec(P_1, i) \neq exec(P_2, i)$.*

Definition 4 (Behavioral Similarity). *The behavioral similarity between two programs P_1 and P_2 that share the same input domain I is $|I_s|/|I|$, where $I_s \subseteq I$, $exec(P_1, I_s) = exec(P_2, I_s)$, and $\forall j \in I \setminus I_s, exec(P_1, j) \neq exec(P_2, j)$,*

Often the time, certain inputs in I are more important than the other inputs. For example, some inputs exercising the main functionality of the program and being used more frequently might be more important than the other inputs. To handle such scenarios, we further extend behavioral similarity as weighted behavioral similarity: the weighted behavioral similarity between P_1 and P_2 is $(\sum_{k=1}^{|I_s|} w_k)/(\sum_{n=1}^{|I|} w_n)$, where $\forall i_n \in I$, w_n is the weight for i_n.

We further define a partial order relation, subsumption, over two programs with respect to a reference program. The relation can be used to determine which program is more similar to the reference program.

Definition 5 (Behavioral Subsumption). *Assuming two programs P_1, P_2 and a reference program P_r all share the same input domain I, P_2 behaviorally subsumes P_1 with respect to P_r, denoted as ($P_1 \leq_{P_r} P_2$), iff $\exists i \in I, exec(P_2, i) = exec(P_r, i) \wedge exec(P_1, i) \neq exec(P_r, i)$, and $\nexists j \in I, exec(P_1, j) = exec(P_r, j) \wedge exec(P_2, j) \neq exec(P_r, j)$.*

Intuitively, a program P_2 *behaviorally subsumes* a program P_1 if and only if the set of inputs correctly handled by P_1 is a proper subset of that correctly handled by P_2. An input is correctly handled by a program P_i with respect to the reference program P_r if and only if P_i and P_r produce the same output on this input. Figure 2 shows an example illustrating behavioral similarity and subsumption. The reference program is shown in Figure 2a, and the programs in Figures 2b and 2c are incomplete in the sense that they produce the correct outputs for only a subset of the input domain. The three programs behave the same on the inputs satisfying Conditional 1, but behave differently on inputs not satisfying Conditional 1. Moreover, all inputs correctly handled by Program 1 are also correctly handled by Program 2, but not vice versa because Program 2 correctly handles the input value 383 while Program 1 does not. Thus, Program 2 makes more progress towards the reference program than Program 1. By our definition, we say that Program 2 behaviorally subsumes Program 1 with respect to the reference program. In other words, Program 2 is behaviorally more similar to the reference program than Program 1 is.

```
public int Reference(int x) {
    if (x % 7 == 0) return 8; // Conditional 1
        return x % 7; // Conditional 2


}
```

(a) Reference program

```
public int Program1(int x) {
    if(x % 7 == 0) return 8;
    if(x == 1009) return 1;
        return x;

}
```

(b) Program 1

```
public int Program2(int x) {
    if(x % 7 == 0) return 8;
    if(x == 1009) return 1;
    if(x == 383) return 5;
        return x;
}
```

(c) Program 2

Figure 2: An example of behavioral subsumption

4. METRICS FOR BEHAVIORAL SIMILARITY

To compute the precise behavioral similarity between two programs defined in Section 3, a naive approach is to run two programs on every input in the input domain, compare the outputs of the two programs, and then compute the portion of the agreed inputs. Such approach is impractical and infeasible for domains with infinite inputs. In fact, there are many programs, even very simple ones, whose input domain is infinite. For example, any program taking an unrestricted string as the input has an infinite input domain because the length of the string could be arbitrarily long. Enumerating every input for these programs is simply impossible. Even when the input domain is finite, in most cases, it is still impractical to compute the behavioral similarity by definition due to the sheer size of the input domain. A program taking one 32-bit integer as the input has an input domain of size 2^{32}. If there are other parameters, the size of the input domain would be the product of the input domain size of each parameter, blowing up very quickly.

To address these issues, we make a trade-off between the precision and the cost of computing behavioral similarity. We propose three practical metrics that compute approximate behavioral similarity between two programs. The goal of these metrics is to provide highly accurate approximations to the actual behavioral similarity at a low cost. We next describe the three metrics in detail. For each metric, we describe why we propose this metric, how to compute it, and the strength and weakness of this metric.

4.1 Random Sampling (RS)

Since computing the behavioral similarity via enumerating all possible inputs from the input domain is prohibitively expensive or infeasible, we instead use the randomly sampled inputs to approximate the computation, thus reducing the cost. The insight behind RS is that by uniformly sampling a significant portion of inputs from the input domain, the behavioral similarity computed based on the sampled inputs can provide an accurate approximation to the actual behavioral similarity. To compute the value for RS, we first use random test generation to generate inputs uniformly distributed over the input domain. Then we run both programs on each test input separately to compare their outputs. The proportion of the agreed inputs over the sampled inputs is the value for RS. The formal definition of RS is as follows:

Definition 6 (RS). *P_r and P_c are two programs sharing the same input domain I. Let I_s be a set of inputs randomly sampled from I, and I_a be a subset of I_s such that $\forall i \in I_a, exec(P_r, i) = exec(P_c, i)$ and $\forall j \in I_s \setminus I_a, exec(P_r, j) \neq exec(P_c, j)$. The RS metric is defined as $M_{RS}(P_r, P_c) = |I_a|/|I_s|$.*

RS is a straightforward yet effective and efficient (i.e., low cost) metric for approximating behavioral similarity. When the input domain is huge, or even infinite, RS can always sample an affordable portion of inputs to compute a reasonably good approximation to the actual behavioral similarity. Since RS treats the program as a black box, and does not analyze the program for test generation, the cost of generating test inputs and measuring RS is rather low or even negligible. On the other hand, due to the black-box test generation, it may miss some inputs (e.g., corner cases) that reveal small behavioral differences between programs, and thus cannot distinguish slightly different programs.

4.2 Single-program Symbolic Execution (SSE)

The Single-program Symbolic Execution (SSE) metric approximates behavioral similarity based on the insight that the number of program paths is typically much smaller than the size of the input domain. The program paths can be viewed as a more succinct representation of program behaviors. If we pick one input for each path in the program, then these inputs explore the representative behaviors of the program. Hence, it is less costly to measure the behavioral similarity by running the program under analysis on only the representative inputs of the reference program and comparing their outputs. Thanks to the recent advance of DSE, we leverage DSE techniques to efficiently explore program paths. To compute SSE, we choose one program as the reference program, and apply DSE to generate test inputs that capture the behaviors of the reference program. We then run the other program on these test inputs and compute the proportion of agreed inputs over the generated inputs as the value of SSE. The formal definition of SSE is as follows:

Definition 7 (SSE). *P_r and P_c are two programs sharing the same input domain I, and P_r is the reference program. Let I_s be the set of inputs generated by DSE on P_r, and I_s be a subset of I_s such that $\forall i \in I_a, exec(P_r, i) = exec(P_c, i)$ and $\forall j \in I_s \setminus I_a, exec(P_r, j) \neq exec(P_c, j)$. The SSE metric is defined as $M_{SSE}(P_r, P_c) = |I_a|/|I_s|$.*

In contrast to RS, SSE explores different feasible paths to generate test inputs. Thus, these test inputs are more likely to cover those corner cases of the program. Revealing such corner cases is helpful in distinguishing programs with small behavioral differences. But still, SSE has some limitations. First, SSE never considers the program under analysis, but generates test inputs based on only the reference program. The generated test inputs do not necessarily capture all behaviors of the program under analysis. It is possible that all the generated test inputs agree on both programs, but the program under analysis still has some additional behaviors (not revealed by the test inputs) that are different from those of the reference program. In such

```
public void PairedProgram (object[] args) {
    Debug.Assert(Program1(args) == Program2(args));
}
```

Figure 3: The template of paired programs

cases, SSE would incorrectly consider these two programs to be behaviorally equivalent. Second, programs may have infinitely many paths when there are loops in the program whose iteration count depends on unbounded inputs [30]. For these programs, it is also infeasible to enumerate all program paths. To alleviate this issue, we can either bound the input domain or the loop iteration count to enumerate a subset of program paths as an approximation.

4.3 Paired-program Symbolic Execution (PSE)

To address the limitation that SSE may fail to reveal behaviors in the program under analysis, our Paired-program Symbolic Execution (PSE) metric is computed by constructing a paired program [25] from the reference program and the program under analysis, and generating test inputs by exploring the paths in the paired program. Figure 3 shows the template for constructing the paired program. The paired program shares the same input domain with the two programs, and it feeds the same input to both programs and asserts the outputs of the two programs to be the same. When generating test inputs based on the paired program, DSE attempts to generate test inputs passing and failing the assertion, respectively. The inputs passing the assertion indicate that the two programs produce the same output on these inputs, while the inputs failing the assertion indicate that the two programs produce different outputs. Hence, we compute the proportion of test inputs that pass the assertion as the value of PSE. The definition of PSE is as follows:

Definition 8 (PSE). *Given two programs P_r and P_c sharing the same input domain I, we construct the paired program P_p in the form of $assert(exec(P_r, I) = exec(P_c, I))$, where assert is a function that accepts a condition as input and asserts that the condition is true. Let $exec(P_p, i) = \top$ denote the execution of an input i on P_p that passes the assertion. Let I_s be the set of inputs generated by DSE on P_p, and I_a be a subset of I_a such that $\forall i \in I_a, exec(P_p, i) = \top$ and $\nexists j \in I_s \setminus I_a, exec(P_p, j) = \top$. The PSE metric is defined as $M_{PSE}(P_r, P_c) = |I_a|/|I_s|$.*

Since the test generation for PSE is based on the paired program constructed from both programs, PSE improves SSE by avoiding the situations where the generated test inputs capture behaviors of the reference program but not the program under analysis. However, PSE still faces the same challenge of handling infinite paths in some paired programs. To alleviate this issue, we can again bound the input domain or loop iteration counts. In addition, the paths to be explored by PSE in the paired program are the combination of paths in both programs. Thus, PSE has a higher cost in path exploration than SSE does.

5. EVALUATIONS

We evaluate the effectiveness of our proposed metrics on code artifacts produced in three real-world data sets from the Pex4Fun platform [1]. Our evaluations intend to answer the following two research questions:

- **RQ1:** How effective are our metrics in ordering programs based on their progress towards the reference program?
- **RQ2:** How accurate are our metrics in approximating the behavioral similarity?

The answer to RQ1 shows the effectiveness of our metrics in determining behavioral equivalence and subsumption between programs. The answer to RQ2 shows how accurately our metrics can be used to quantify behavioral similarity between programs.

Table 1: Evaluation subjects

Subject	#Tasks	#Students	#Submissions
CSharp4Fun	11	80	1724
APCS	156	37	4116
ICSE2011	30	29	6129
Total	197	146	11969

5.1 Subjects and Evaluation Setup

Subjects. Pex4Fun allows teachers to pose a set of programming tasks, each associated with a secret solution program. Students keep submitting their programs until the submitted program behaves the same as the solution program does. We construct three data sets from three real courses on the Pex4Fun platform, namely CSharp4Fun, APCS, and ICSE2011. CSharp4Fun is a short course that introduces the basics of C#, with a set of tasks exercising basic programming concepts. APCS is a companion course with a series of exercises for advanced placement computer science students. ICSE2011 is a set of programming tasks used in the contest of Pex4Fun held at ICSE 2011. These courses contain various programming tasks, ranging from algorithmic problems and object-oriented design problems to testing problems. For each task in a course, we collect both the reference program and a sequence of programs submitted by students. Typically, the size of the submitted programs and the reference programs of these tasks ranges from a few lines to a hundred lines of code. We then filter out those submissions with syntactic errors because our evaluations focus on the behaviors of the programs. Note that one sequence in our data sets corresponds to a sequence of submissions for one task from one student. Table 1 shows the details of our subjects, including the number of tasks, the number of students, and the number of programs submitted by students.

Evaluation Setup. To answer RQ1, we randomly sample sequences of programs from our data sets, with each sequence satisfying either of the following properties: (1) the sequence has a later program behaving equivalently as an earlier program; (2) the sequence has a later program behaviorally subsuming an earlier program with respect to the reference program. In other words, the sequence is ordered in that an earlier program is either behaviorally equivalent with or subsumed by a later program. Due to the significant manual efforts required in the selection process, we select 60 such sequences (20 for each data set) for our evaluations.

To measure the effectiveness of each metric in ordering programs based on their progress towards the reference program, we compute all three metric values for each program in each sequence, and verify whether the computed values of a metric over a sequence conform to the sequence order. In other words, two equivalent programs should have the same value, and a program subsuming the other should have a higher value than the other. A metric is marked as correctly indicating the progress over the programs in the sequence if the computed values of the metric conform to the order of the sequence. We use the percentage of sequences on which a metric produces the correct order to show the effectiveness of the metric in indicating the progress over programs. A higher percentage for a metric indicates that the metric is more accurate in indicating the ordering of programs.

To answer RQ2, we select tasks that are feasible for manually computing behavioral similarity between two programs, and then check how accurately our metric values approximate the manually computed behavioral similarity. For RQ2, we also randomly sample 60 sequences (20 for each data set) from these tasks. For each program in the sequences, we manually compute the behavioral similarity between the program and the reference program. To assure the reliability of the manually computed behavioral similarity, we have at least two authors agree on every manually computed result. If the input domain of the programs is infinite, we provide bounds to the input domain. We bound integers within the interval [-2147483648, 2147483647] (i.e., 32-bit integers). We bound the length of a string to be 10, and each character in the string to be from only the ASCII table (i.e., 8-bit characters). We also bound the length of an array to be 10. In addition, we exclude any task that accepts floating-point numbers as inputs because we cannot enumerate floating point numbers even with bounds. We plan to design other means to evaluate our metrics on programs taking as input floating point numbers in future work.

With a bounded finite input domain, for any program P_i in a sequence Q_j, we compute the actual behavioral similarity S_{ij} (manually) between P_i and its reference program as well as the values of each metric M_{ij} on these two programs using our tool. We measure the accuracy of each metric by *absolute error* between the manually computed behavioral similarity and the metric values as follows:

$$e_{ij} = |S_{ij} - M_{ij}|$$

We further compute the average absolute error of all programs in each subject data set and its standard deviation to show the overall effectiveness of our metrics in approximating behavioral similarity.

5.2 RQ1: Effectiveness of Indicating Progress

Table 2 shows the summary of our evaluation results on RQ1. Column "Subject" shows the subject data sets. Column "#Seq." shows the number of program sequences selected from each data set. Column "#Tasks" shows the number of distinct tasks included in the selected sequences. Column "#Avg. Length" shows the mean number of programs contained in each sequence. Columns "#CO of RS", "#CO of SSE", and "#CO of PSE" show the number of sequences on which RS, SSE, and PSE produce the correct order based on behavioral similarity, respectively.

As shown in the table, PSE performs the best in ordering the sequences. For 52 out of 60 sequences (87%), it successfully produces the correct order. The reason for its good performance is that it does path exploration on the paired program, combining both the reference program and the submitted program. Thus, PSE is sensitive to changes in the submitted program. In general, there are two kinds of changes in the submitted program that can be effectively detected by PSE: (1) control-flow changes that result in different or additional paths in the paired program; (2) non-control-flow changes that make previously infeasible paths in the paired program feasible. PSE can detect these changes because the path exploration on these two paired programs (before and after changes) identifies two different sets of paths. The metric values computed based on the path exploration are then different. If such changes in the submitted program correctly handle more inputs, PSE could typically generate more passing test inputs and thus increase the metric value. In this way, PSE effectively indicates the progress between programs. The other two metrics, RS and SSE, do not work well on producing the correct orders over the sequences, because they are not sensitive to the changes of the submitted program (not generating test inputs based on the submitted program). In other words, given a reference program, they both run any submitted program on a fixed set of test inputs. Hence, they often fail to distinguish two submitted programs with slight behavioral differences.

Figure 4 shows a simple example from our data sets, where PSE successfully indicates the progress between programs. Figures 4a, 4b, 4c, and 4d respectively show the reference program, two submitted incorrect programs (Submissions 1 and 2), and a template of paired programs constructed by PSE. It can be observed that Submission 1 correctly handles exactly one input (0), and Submission 2 correctly handles exactly two inputs (0 and 42). By the definition of *Behavioral Subsumption*, Submission 1 is behaviorally subsumed by Submission 2 with respect to the reference program (i.e., Submission 2 makes progress over Submission 1). Although the behavioral differences between these two submissions are very small (on only one input), PSE can still correctly indicate the progress because there are control-flow changes from Submission 1 to Submission 2. PSE constructs two paired programs PP_1 and PP_2 by replacing the Submission method in the template with Submissions 1 and 2, respectively. PSE then explores paths in both PP_1 and PP_2, and finds that there are two feasible paths in PP_1, one of which leads to the violation of the assertion. Similarly, PSE finds that one out of the three feasible paths in PP_2 leads to the violation of the assertion. Thus, the metric values of Submissions 1 and 2 are $0.5 = 1/2$ and $0.67 = 2/3$, which correctly reflect the order between Submissions 1 and 2. However, both RS and SSE fail on the example in Figure 4. In order to distinguish Submissions 1 and 2, they need to generate the test input $x = 42$. The probability that RS generates 42 for x is $1/2^{32}$, which is close to 0. For SSE, since it generates test inputs based on only the reference program, it generates only one input $x = 0$ (the default value for integers) in this case because there is only one path in the reference program.

There are in total 8 sequences on which PSE produces wrong orders, corresponding to 8 failures. We investigate these wrong orders and identify two major reasons. For 3 out of the 8 failures, PSE fails to detect the behavioral differences because the changes do not modify the control flow of the paired program. Consider an extreme example where the reference program takes as input an integer x and always returns 1 regardless of the input. Suppose that an earlier submission P_1 simply returns x, and a later submission P_2

Table 2: Results on ordering programs based on behavioral similarity

Subject	#Seq.	#Tasks	#Avg. Length	#CO of RS	#CO of SSE	#CO of PSE
CSharp4Fun	20	5	3.5	15 (75%)	10 (50%)	18 (90%)
APCS	20	17	4.2	11 (55%)	9 (45%)	17 (85%)
ICSE2011	20	7	4.7	10 (50%)	7 (35%)	17 (85%)
Total	60	29	4.1	36 (60%)	26 (43%)	52 (87%)

```
int Reference(int x) {

    return 42 − x;

}
```

(a) Reference Program

```
int Submission1(int x) {

    return 42;

}
```

(b) Submission 1

```
int Submission2(int x) {
    if (x == 0) return 42;
    if (x == 42) return 0;
    return 0;
}
```

(c) Submission 2

```
void PairedProgram(int x) {
    Debug.Assert
        (Reference(x) ==
        Submission(x));
}
```

(d) Paired Program

Figure 4: A simple example where PSE works but neither RS nor SSE works

returns x^2 instead. It is easy to see that neither programs are equivalent to the reference program, but P_2 subsumes P_1 because P_2 correctly handles the input -1 in addition to the input 1. However, when computing the metric value of PSE, both paired programs for P_1 and P_2 have two paths, and one of the paths violates the assertion. Hence, PSE generates two test inputs exercising both paths for each paired program, and gives the same metric value of 0.5 to both P_1 and P_2. Essentially, since the control flow of the paired program and the feasibility of each path remain the same, the percentage of the generated passing test inputs remains the same. The other major reason (for 4 out of the 8 failures) is that the control-flow changes in the paired program actually lower the percentage of passing test inputs. In general, the changes allow the current program to correctly handle additional inputs, the metric value should increase. However, in some corner cases, the control-flow changes affect the path exploration of Pex and lower the percentage of the passing test inputs. For instance, although Pex generates more passing test inputs, it may generate even more failing test inputs, and thus makes the metric value decrease.

5.3 RQ2: Effectiveness of Quantifying Behavioral Similarity

Table 3 shows the summary of our evaluation results of RQ2. Column "Subject" shows the subject data sets. Column "#Seq." shows the number of submission sequences selected from each data set. Column "#Tasks" shows the number of distinct tasks included in the selected sequences. Column "#Avg. Length" shows the mean number of programs contained in each sequence. Columns "AE/SD of RS", "AE/SD of SSE", and "AE/SD of PSE" represent the average absolute errors and their standard deviation for RS, SSE, and PSE, respectively.

As shown in Table 3, RS approximates the actual behavioral similarity much better than the other two metrics. Its overall average absolute error (0.017) and standard deviation (0.044) are very close to 0. The reason for its accurate approximation is that RS ensures that the randomly generated inputs are uniformly distributed across the input domain. Consider the input domain I as two big partitions, one partition I_a containing all agreed inputs, and the other partition I_d containing all non-agreed inputs. The behavioral similarity is computed as $|I_a|/|I|$. Since RS uniformly samples inputs, the possibility of sampling an agreed input is also $|I_a|/|I|$. Hence, when RS samples a significant number of inputs, the percentage of the sampled agreed inputs (the metric value for RS) would be very close to $|I_a|/|I|$.

SSE and PSE do not approximate the behavioral similarity well because they compute the metric values based on only path exploration. They implicitly give an equal weight to each path by doing simple path counting. However, the corresponding input partition of each path may vary in size. In order to provide a more accurate approximation, we would need to assign a weight to each path based on the size of its corresponding input partition. In future work, we plan to investigate how the probability of executing a path [10] can be used to improve SSE and PSE.

It is interesting to observe that, in some cases, SSE and PSE provide extremely accurate approximations (e.g., produce perfect values for behavioral similarity). In such cases, the input domain of the program is typically very small (possible for the DSE engine to enumerate within its time bound), and each input leads to a unique path in the program. When DSE exhaustively explores all the paths in the program, it actually enumerates all the inputs from the input domain. The generated test inputs are the set of all possible inputs. Hence, the metric values computed for SSE and PSE are exactly the actual behavioral similarity. Figure 5 shows such an example from our data sets where both SSE and PSE produce perfect metric values for behavioral similarity. The program in Figure 5a is the reference program, and the programs in Figures 5b and 5c are an incorrect submission and a correct submission, respectively. It can be seen that every input of the reference program leads to a unique path. Since the value range of a byte argument is from 0 to 255, there are in total 256 paths in the reference program. When generating test inputs based on the reference program, SSE can easily explore all the paths and generate 256 test inputs, consisting of every byte value from 0 to 255. Similarly, the paired programs for Submissions 1 and 2 also have 256 paths. PSE exhaustively explores all the 256 paths and generates the same test suite as SSE does. Since the generated test inputs for both SSE and PSE are the set of all possible inputs, the computed metric values are exactly the actual behavioral similarity.

The cost of computing the three metrics is acceptable. As expected, RS has lower cost than the other two metrics. The major cost for computing RS is running the sampled inputs on two programs. Since we focus on introductory programs, the time for running one input on these programs ranges from milliseconds to a second. The cost of computing RS for a pair of programs is typically less than one minute (in many cases, just a few seconds). We can further reduce the cost of RS by controlling the size of the sample set. For SSE and PSE, the major cost is input generation. The cost of generating inputs includes both the analysis cost (e.g., symbolic execution, constraint solving) and the execution cost (e.g., concrete execution). Since DSE actually performs the concrete execution, we get not only the inputs but also

Table 3: Results on quantifying behavior similarity

Subject	#Seq.	#Tasks	#Avg. Length	AE/SD of RS	AE/SD of SSE	AE/SD of PSE
CSharp4Fun	20	5	3.5	0.020 / 0.031	0.184 / 0.224	0.085 / 0.158
APCS	20	17	4.2	0.027 / 0.072	0.225 / 0.309	0.177 / 0.141
ICSE2011	20	7	5.1	0.007 / 0.014	0.390 / 0.456	0.293 / 0.270
Total	60	29	4.3	0.017 / 0.044	0.289 / 0.374	0.202 / 0.240

```
public static int Puzzle(byte number) {
    int ct = 0;
    while (number != 0) {
    if (number % 2 > 0) ct++;
        number >>= 1;
    }
    return ct;
}
```

(a) Reference Program

```
public static int Puzzle(byte number) {


    return (int) (number / 2);



}
```

(b) Submission 1

```
public static int Puzzle(byte number){
    var sum = 0;
    for (int i = 0; i < 8; i++)
    {
        sum += (number >> i & 1);
    }
    return sum;
}
```

(c) Submission 2

Figure 5: An example from our data sets where SSE and PSE produce perfect metric values for behavioral similarity

their outputs after the input generation. Thus, we do not need to run the generated inputs again to obtain the outputs. We set a two-minute time bound for the input generation of a program. Thus, the cost of computing SSE and PSE is at most slightly over two minutes. Typically, the cost of computing SSE and PSE for a pair of programs ranges from ten seconds to one minute.

6. APPLICATIONS OF BEHAVIORAL SIMILARITY

There are broad applications of our behavioral similarity metrics for improving online programming and software engineering education. We next discuss three practical applications in detail.

Progress Indication. Our metrics can be used as a progress indicator during students' coding process. We can constantly compute metric values in the background while students are coding, and display the metric values in real time as a progress indicator. Note that we do not compute metric values for the code with syntactic errors, as such code cannot be run. We believe that such an indicator is useful to students so that they are able to keep track of their progress. Generally, if they observe increasing metric values, they know that they are moving along the right direction. Thus, they may feel more confident in their coding. If they observe a significant decrease, it is very likely that they are doing something wrong. They may stop early and avoid making further mistakes.

As shown in our evaluations, PSE is most effective in distinguishing small behavioral differences while RS is very effective in approximating the actual behavioral similarity. We can combine these two metrics to produce a more informative progress indicator. In particular, we can use PSE to indicate *whether* there is any progress and RS to indicate *how substantial* the progress is. Consider the example in Figure 2. Although Program 2 makes progress over Program 1, the progress is small and Program 2 actually moves along a wrong direction towards the reference program (overfitting some specific inputs). If we compute PSE for the two programs against the reference program, the result values are 0.67 and 0.80 for Programs 1 and 2, respectively, indicating that Program 2 is making substantial progress over Program 1. Similarly, if we also compute the metric values of RS, the result values are the same as 0.13 for both programs, indicating that Program 2 does not make any progress over Program 1. Thus, neither PSE nor RS alone provides accurate information on the actual progress. However, if we combine these two metrics as suggested, we can come to the correct conclusion that although Program 2 is making progress over Program 1 (Program 2 has a higher metric value of PSE), the progress is not substantial (Programs 1 and 2 have the same metric value for RS).

Automatic Grading. To reduce the grading burden on instructors, we can use our metrics to improve syntactical approaches [2,24] to automatic grading. The behavioral similarity between a student's program and the reference program can be a factor in grading. Higher similarity generally indicates a higher grade. Since our metrics are purely based on program semantics, we can combine our metrics with other syntactical approaches [2, 24] to produce a better automatic grader. The syntactical approaches address the limitations of our metrics in cases where students' programs are syntactically very similar to the reference program but semantically quite different (e.g., missing sanity check on a large input domain). Our metrics address the limitations of the syntactical approaches in cases where students' programs are semantically very similar to the reference program but syntactically quite different (e.g., the programs in Figure 1). There are various ways of integrating syntactic and semantic approaches to produce the final grade. For instance, we can use a sum of weighted behavioral and syntactic similarities as the final grade. Alternatively, we can also use the maximum value of behavioral and syntactic similarities as the final grade.

In addition, our metrics also provide the flexibility of adding weights to the input. Typically, in practice, not every input is of the same importance. Some inputs that exercise the main functionality of the task may carry higher weights while some other inputs filtered by sanity checks may carry lower weights. Our *weighted* behavioral similarity allows instructors to easily specify the weights over the input domain.

Hint Generation. We can also use our metrics to generate hints for students during their coding process. Since students cannot always get prompt help from others, it is beneficial to automatically generate hints based on their current code when they encounter problems. The idea, inspired by crowdsourcing, is to leverage our metrics to find behaviorally similar programs from other students as hints. Specifically, suppose that a student S_i needs help on their current program P_i. We can first compute the behavioral similarity M_i between P_i and the reference program P_r using RS. Then we search among other students' submissions and find one submission P_j^k (from a student S_j who has successfully completed the task), whose similarity M_j^k against P_r is equal or closest to M_i. k denotes the k^{th} submission of a submission sequence. Next, we further look into the submission sequence of the student S_j, and select as a hint program the first submission P_j^n such that $k < n$ and $M_j^k < M_j^n$. Intuitively, the hint program P_j^n contains the *immediate* progress made over the program P_j^k. Since P_i is similar to P_j^k, by looking at P_j^n, the student S_i may get a hint on the next step towards the solution. We may even use DSE to generate some hint inputs, which produce different outputs on P_i and P_j^n, so that the student knows what inputs to handle in the next step. Note that it is desirable to combine this technique with syntactical analysis. When searching for P_j^k, it is also important to find a syntactically similar program. If P_j^k is syntactically very different from P_i, the student may need to spend more time understanding the hint program. More importantly, the next step may not be applicable to P_i.

7. THREATS TO VALIDITY

Threats to external validity primarily lie in the generality of the data sets used in our evaluations. All three data sets are collected from the PEX4FUN platform, and all programs are written in C# only. Although such data sets are representative for the PEX4FUN platform and object-oriented languages, they may not be representative for other platforms (e.g., MOOC) and programing languages (e.g., scripting and functional languages). In future work, we plan to evaluate our metrics on data sets from various platforms, and also develop metrics of behavioral similarity for other types of programming languages.

Threats to internal validity are mostly potential human errors in our evaluations. There is a significant amount of manual effort required to accurately determine the subsumption relations for answering RQ1. In addition, the manual computation of behavioral similarity used in the evaluation of RQ2 may be complex and error-prone. These threats are mitigated by double-checking all the manual work. The results are individually verified and cross-checked by at least two authors. There may also be some faults in our software evaluation framework, and these faults could affect the validity of our results. This threat is mitigated by extensively testing the system before conducting the evaluations.

8. RELATED WORK

Automatic Grading. Alur et al. [2] propose an approach that compares students' incorrect deterministic finite automatons (DFAs) with the reference DFA. Their approach assigns partial grades using techniques that capture both syntactic differences based on syntactic edit distance and semantic differences based on the accepted string inputs. Although both their approach and ours use similar notions for semantic similarity, their approach works on DFAs while our metrics work on programs.

Singh et al. [24] propose an approach that automatically determines the minimal fixes to make a student's incorrect solution match the behavior of a reference solution. Their approach focuses on providing feedback on how to fix the incorrect solutions via computing the minimal syntactic distance to the reference solution, while our metrics focus on quantifying how similar the semantics of two programs are based on input/output behaviors. Our metrics can identify similar programs with different syntactic structures.

There also exists previous work leveraging the semantic similarity of students' programs and reference programs [29]. Such approach performs semantic-preserving transformation to standardize both the students' programs and the reference programs, and compares their system dependence graphs to compute similarities. Instead of comparing system dependence graphs, our approach compares input/output pairs of programs for computing behavioral similarities, and thus can identify similar programs with different system dependence graphs.

Equivalence Checking. There exist several approaches that check semantic/behavioral equivalence of programs using program dependence graphs [3, 5], input-output dependence [14], symbolic summaries [22], and side-effect summaries [19]. All these approaches yield a boolean answer to the equivalence checking, and some of them also produce the behavioral delta. However, in addition to equivalence checking, our approach also quantifies how similar two programs are. To the best of our knowledge, we are the first to propose an approach for quantifying the behavioral similarity of programs.

Jiang et al. [16] propose an approach to identify functionally equivalent code fragments via random testing. Their approach considers two code fragments equivalent based on input/output behaviors rather than syntactic structures. Our RS metric is similar to their approach in regard to using random testing. However, we propose two additional metrics, which use DSE-based test generation and outperform the RS metric in detecting small variations between code fragments.

Test-Case-Based Feedback. Hext et al. [13] propose an approach that automatically grades simple program exercises by comparing the stored data with the data generated by executing incorrect programs. Jackson et al. [15] propose an approach that automatically checks the correctness of programs and programming styles such as modularity, complexity, and efficiency. PEX4FUN uses DSE to generate test inputs that cause the incorrect programs and the correct program to produce different outputs. All of these approaches produce feedback by showing the test inputs of the failing test cases, while our metrics measure the proportion of inputs producing the same output on the students' program and the reference program, and can be used for progress indication and hint generation.

Code-Clone Detection. Researchers have proposed approaches that compute similarities of various representations of code fragments to automatically identify code clones, such as tokenized statements [17], abstract syntax trees [4], program dependence graphs [18], and metrics based on syntactic units (e.g., classes and functions) [8, 20]. These clone-detection approaches focus on fragments of source code and compute similarities based on the syntactic or semantic representations of code fragments. Unlike these static approaches that compute similarities by statically analyzing code artifacts, ours is a dynamic approach that computes similarities by running programs to produce input/output pairs.

9. CONCLUSION

In this paper, we have presented three metrics, namely RS, SSE, and PSE, to approximate the behavioral similarity between programs. We leverage test-generation techniques to generate inputs and compute metric values by running the inputs on the programs. We have implemented our metrics based on Pex and evaluated them on three data sets from the Pex4Fun platform. The evaluation results show that RS provides the best approximation to the behavioral similarity with an average absolute error of 0.017, but may not be able to indicate small progress over programs. As a complement, PSE is effective (correct in 87% cases) in ordering programs based on behavioral similarity, but does not provide as good approximations as RS does. In future work, we plan to integrate these two metrics into one metric that provides both good approximation and sensitivity. We have also demonstrated a number of applications of our metrics for online programming and software engineering education including progress indication, automatic grading, and hint generation.

Acknowledgment

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. This work is also supported in part by NSF grants CCF-1349666, CCF-1409423, CNS-1434582, CCF-1434596, and CNS-1513939, and a Google Faculty Research Award.

10. REFERENCES

[1] Pex4Fun. http://www.pex4fun.com/.

[2] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *Proc. IJCAI*, pages 1976–1982, 2013.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. POPL*, pages 384–396, 1993.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.

[5] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proc. ICSM*, pages 41–50, 1992.

[6] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux. Code Hunt: Experience with coding contests at scale. *Proc. ICSE, JSEET*, pages 398–407, 2015.

[7] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, 2010.

[8] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: Tuning code clones at hands of engineers in practice. In *Proc. ACSAC*, pages 369–378, 2012.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[10] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proc. ISSTA*, pages 166–176, 2012.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, pages 151–166, 2008.

[13] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.

[14] D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *Proc. ICSM*, pages 243–252, 1994.

[15] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proc. SIGCSE*, pages 335–339, 1997.

[16] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. ISSTA*, pages 81–92, 2009.

[17] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[18] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. SAS*, pages 40–56, 2001.

[19] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. CAV*, pages 712–717, 2012.

[20] E. Merlo, G. Antoniol, M. Di Penta, and V. F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *Proc. ICSM*, pages 412–416, 2004.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[22] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[23] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[24] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proc. PLDI*, pages 15–26, 2013.

[25] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.

[26] N. Tillmann and J. de Halleux. Pex–White box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[27] N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *Proc. ASE*, pages 385–396, 2014.

[28] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *Proc. ICSE, SEE*, pages 1117–1126, 2013.

[29] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. *Inf. Softw. Technol.*, 49(2):99–107, 2007.

[30] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. ASE*, pages 246–256, 2013.