PaCon: A Symbolic Analysis Approach for Tactic-Oriented Clustering of Programming Submissions

Yingjie Fu
yingjiefu@stu.pku.edu.cn
Peking University
Beijing, China

Jonathan Osei-Owusu
jo28@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

Angello Astorga
aastorg2@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

Zirui Neil Zhao
ziruiz6@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

Wei Zhang
zhangw.sei@pku.edu.cn
Peking University
Beijing, China

Tao Xie
taoxie@pku.edu.cn
Peking University
Beijing, China

Abstract

Enrollment in programming courses increasingly surges. To maintain the quality of education in programming courses, instructors need ways to understand the performance of students and give feedback accordingly at scale. For example, it is important for instructors to identify different problem-solving ways (named as *tactics* in this paper) used in programming submissions. However, because there exist many abstraction levels of tactics and high implementation diversity of the same tactic, it is challenging and time-consuming for instructors to manually tackle the task of *tactic identification*. Toward this task, we propose *PaCon*, a symbolic analysis approach for clustering functionally correct programming submissions to provide a way of identifying tactics. In particular, *PaCon* clusters submissions according to *path conditions*, a semantic feature of programs. Because of the focus on program semantics, *PaCon* does not struggle with the issue of an excessive number of clusters caused by subtle syntactic differences between submissions. Our experimental results on real-world data sets show that *PaCon* can produce a reasonable number of clusters each of which effectively groups together those submissions with high syntax diversity while sharing equivalent path-condition-based semantics, providing a promising way toward identifying tactics.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Theory of computation** → **Program analysis**.

Keywords: programming education, clustering, symbolic analysis, path condition

ACM Reference Format:
Yingjie Fu, Jonathan Osei-Owusu, Angello Astorga, Zirui Neil Zhao, Wei Zhang, and Tao Xie. 2021. PaCon: A Symbolic Analysis Approach for Tactic-Oriented Clustering of Programming Submissions. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3484272.3484963

1 Introduction

In recent years, programming courses, especially online programming courses [19], have attracted high enrollment from all over the world. As the size of classrooms increases, inspecting programming submissions for grading, constructing reference solutions, and preparing targeted teaching materials has become increasingly time-consuming. Some automated tools [1, 7, 9, 11, 21, 23, 27, 29] have been proposed to assist instructors of online programming courses. These tools typically focus on the *functional correctness* of submissions or how to modify incorrect submissions into functionally correct ones.

However, it is also important to identify the ways (named as *tactics* in this paper) that students solve a problem, for three main reasons. First, identifying tactics can assist instructors in achieving more efficient grading when tactic-related constraints appear in the description of assignments.

* Tao Xie is also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China, and is the corresponding author.

SPLASH-E '21, October 20, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9089-7/21/10…$15.00
https://doi.org/10.1145/3484272.3484963

For example, given the assignment "*find the largest difference between any two elements in an array without sorting*", a submission that first sorts the input array elements and then subtracts the smallest element from the largest element should not receive full marks, because it violates the requirement "without sorting." Because tactic-related characteristics of submissions cannot be reflected by the output of programs, without tool assistance, instructors have to manually inspect each submission to check whether these tactic-related constraints are satisfied. Second, identifying tactics can help instructors efficiently come up with a variety of reference solutions that can be valuable for semi-automated tools to generate customized feedback. Some existing tools [5, 8, 13] for feedback generation require users to provide templates or model solutions to match against programming submissions, but it is challenging for an inexperienced instructor to list comprehensive templates ahead of time. By identifying different tactics used in functionally correct submissions from students, instructors can craft reference solutions with less effort. Third, identifying tactics can help instructors prepare targeted and adaptive teaching materials. For example, when students are asked to *"find the k-th largest number of the given array"*, a good way to solve the problem in this assignment is the way of using "divide and conquer", but many students may use the way of sorting to solve this problem. By identifying the used tactics, instructors can have a view of the distribution of tactics in submissions from students, and then lecture to the class about the relationship of the "sorting" and "divide and conquer" ways.

Identifying tactics in programming submissions is a challenging and time-consuming task for two main reasons. First, it is difficult to provide a clear definition of tactic, because the identification of tactics reflects different levels of abstraction, depending on the specific assignments and teaching requirements. For example, when we focus on the assignment of sorting, different tactics of submissions can be divided into known tactics such as *bubble sort*, *insertion sort*, and *merge sort*. But for the assignment of "*finding the k-th largest elements in an array*", the tactics can be divided into *divide and conquer*, *sorting*, and *brute force*. In the latter assignment, a proper clear definition of tactic ought not to capture the specific used sorting algorithm. Second, programs that adopt the same tactic may have distinct ways of syntactically organizing the code. For example, given the problem of computing C_m^n, both loop and recursion can achieve the same tactic "*computing C_m^n by calculating factorials*", but they show apparent syntactic differences.

To help identify tactics in programming submissions, in this paper, we propose *PaCon*, a symbolic analysis approach for clustering functionally correct programming submissions to provide a way of identifying tactics. *PaCon* uses path conditions from symbolic execution [14] to cluster submissions. Executing a program with the given input value can exercise (i.e., follow) a path in the program. The path condition for this path, also referred to as the path condition for the given input value, represents the input constraints (1) that all input values exercising this path must satisfy and (2) whose satisfying input values must cause to exercise this path.

Our insight underlying *PaCon* is that the input-space-partitioning strategy adopted by a submission can offer a way to look for tactics, and such strategy can be derived based on the path conditions of input values. In particular, path conditions reflect how the program of the submission divides the input space into multiple equivalence classes, each of which includes all the input values that follow the same path. Revisit the earlier described example: two submissions that compute C_m^n by calculating factorials, including one using loops for the calculation and the other one using recursion. Although the two submissions are implemented differently, their programs share the same input-space-partitioning strategy derived based on the path conditions.

Based on this insight, the workflow of *PaCon* consists of three main steps: (1) test generation, (2) path condition collection, and (3) path-condition-based clustering. *PaCon* first runs a structural test generator to generate test inputs (in short as tests), normalizes the set of the generated tests, and then collects the path conditions of these tests. In the last step of clustering, *PaCon* ensures that submissions in the same cluster must have semantically equivalent path conditions for each of the generated tests.

We conduct an evaluation on real-world data sets to show that the number of clusters produced by *PaCon* is reasonable, and *PaCon* is able to produce clusters each of which effectively groups together those submissions with high syntax diversity while sharing equivalent path-condition-based semantics. By manually looking into the clusters produced by *PaCon*, we find that the clusters can be a good indicator of how the submissions differ in their ways to solve the target problem. These results demonstrate that *PaCon* can provide a promising way toward identifying tactics.

This paper makes the following main contributions:

- We raise the awareness of identifying tactics in student programming submissions.
- We propose a symbolic analysis approach named *PaCon* for clustering functionally correct programming submissions to provide a way of identifying tactics.
- We conduct an evaluation on real-world data sets for assessing *PaCon* in clustering functionally correct programming submissions.

2 Motivating Example

In this section, we present a motivating example to illustrate (1) the challenge faced by clustering approaches to help identify tactics and (2) why existing approaches fail in this task. The motivating example shown in Listings 1-3 is about a programming assignment of "*finding the largest difference between any two elements in an array of int values*" where the

input a is not null. In Listings 1-3, max-difference(A), max-difference(B), and max-difference(C) show the snapshots of three submissions, which are all functionally correct.

Listing 1. max-difference(A)

```
// Tactic: compute max and min (using
    built-in API methods) and return their
     difference.
public class Program {
  public static int Puzzle(int[] a) {
    return a.Max()-a.Min();
  }
}
```

Listing 2. max-difference(B)

```
// Tactic: compute max and min (not using
    built-in API methods) and return their
     difference.
public class Program{
  public static int Puzzle(int[] a){
    int max = a[0], min = a[0];
    for (int i = 1;i < a.Length;i++){
      if (a[i] > max){
        max = a[i];
      }
    }
    for (int i = 1;i < a.Length;i++){
      if (a[i] < min){
        min = a[i];
      }
    }
    return max - min;
  }
}
```

Listing 3. max-difference(C)

```
// Tactic: compute all the pairwise
    element differences, then return the
    maximum.
public class Program {
  public static int Puzzle(int[] a) {
    int max=0;
    foreach(int i in a){
      foreach(int j in a){
        max=i-j>max?i-j:max;
      }
    }
    return max;
  }
}
```

Among these three submissions, max-difference(A) and max-difference(B) share similarities in the way that they solve the problem. Specifically, the first two submissions first find the maximum and the minimum elements of the array and then subtract the minimum element from the maximum element to get the expected return. The submission max-difference(C) solves the problem by checking all pairs of elements and comparing the difference.

However, these three submissions also differ in variable names and code structure. In terms of variable names, max-difference(A) and max-difference(B) both use the variable `max` to record the maximum and use the variable `min` to record the minimum, but max-difference(C) directly uses `max` to record the expected maximum difference. In terms of code structure, max-difference(A) uses the built-in `Max` and `Min` API methods, but max-difference(B) uses two for-loops to identify the largest and smallest integer elements in the array, respectively. To compare all the pairwise differences between elements, max-difference(C) implements a two-level nested for-loop.

In this example, if we do not consider the implementation of the `Max` and `Min` API methods, a desirable clustering approach that can help identify tactics should be able to identify the similarities between max-difference(A) and max-difference(B) in their problem-solving ways and group them together. As for the max-difference(C) submission, because it uses a different tactic, a desirable clustering approach should not put it into the same cluster as max-difference(A) and max-difference(B).

Many existing syntax-based approaches rely on specific program features such as control flow, variable sequence, and Abstract Syntax Tree (AST) to cluster submissions. These approaches put two submissions into the same cluster only when their specific program features are the same or highly similar. For example, *CLARA* [9] puts two submissions into the same cluster only if the looping structure of the two submissions match. *OverCode* [6] requires that the sets of statements (after renaming variables) of the two submissions in the same cluster must be the same. These two approaches both put the first two submissions into separate clusters.

From the preceding example, we can see that existing syntax-based approaches do not consider the problem-solving ways of programs, and thus they do not work well in helping identify tactics. Toward identifying tactics, we propose to take semantic program information into account.

3 Approach

In this section, we present our symbolic analysis approach for clustering programming submissions, named *PaCon*, which clusters programming submissions based on *path conditions*, a semantic feature of programs.

3.1 An Overview of *PaCon*

Given a programming problem, *PaCon* takes a set of functionally correct submissions as input and outputs a set of submission clusters. To simplify the subsequent discussion, we assume that each submission has a static method, named *Puzzle*, which takes inputs through its arguments `args` and returns its result through the return value. The types of the arguments and the return can be either primitive types or classes.

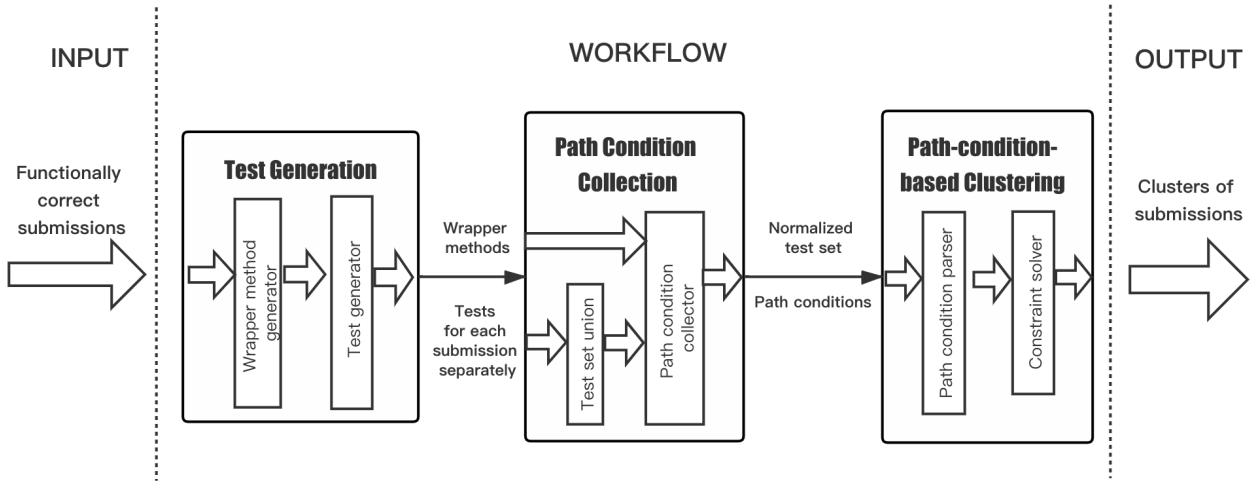

Figure 1. The Workflow of *PaCon*

Figure 1 shows the overview of *PaCon*'s workflow, which consists of three steps: (1) test generation, (2) path condition collection, and (3) path-condition-based clustering. The purpose of the first step is to generate high-code-coverage tests, which can help reflect the program behavior of the submissions. In the second step, for every submission, *PaCon* collects its path condition from running each test on this submission. Finally, *PaCon* groups submissions based on the equivalence of their path-condition semantics and outputs a set of submission clusters.

3.2 Test Generation

The purpose of this step is to attain tests with high code coverage such as high statement, block, branch, data flow, and path coverage. Ideally, the tests should comprehensively sample input values that can help exercise different paths and thus different resulting path conditions, in order to differentiate different input-space-partitioning strategies across submissions.

This step includes two modules: a wrapper method generator and a test generator. *PaCon* first constructs a wrapper method for the programming problem. The wrapper method can be regarded as a parameterized unit test [26] of the *Puzzle* method. The wrapper method can be generated automatically with respect to the argument types and the return type of any functionally correct solution of the problem. Listing 4 presents a simplified version of the wrapper method for *max-difference* shown in Section 2. If there is any precondition (i.e., constraint for the input arguments), instructors can manually add the precondition in the generated wrapper method. Note that only one wrapper is needed for each problem, so it is an acceptable workload for instructors to prepare the wrapper method even manually.

Listing 4. A simplified example of the wrapper method for *max-difference*

```
1 public static void Wrapper(int[] args) {
2   global::Program.Puzzle(args);
3 }
```

After the wrapper method is ready, *PaCon* interacts with a structural test generator to attain tests with high code coverage. Note that it may be impossible to achieve full coverage of paths in a program, e.g., one where the number of loop iterations is the value of an integer input argument without any bound constraint, and thus there exist an infinite number of paths and path conditions. The goal of a structural test generator typically can be configured as achieving high statement, block, branch, or data flow coverage. In our implementation of *PaCon* , the adopted structural test generator is *Pex* [25], whose configuration by default is to generate tests with high block coverage. For the motivating example shown in Section 2, the number of tests generated by *Pex* for max-difference(A), max-difference(B), and max-difference(C) is 1, 6, and 2, respectively.

3.3 Path Condition Collection

In this step, *PaCon* aims to collect the path condition derived from each test. Executing a program with the given input value in the test can exercise (i.e., follow) a path in the program. The path condition for this path, also referred to as the path condition for the given input value, represents the input constraints (1) that all input values exercising this path must satisfy and (2) whose satisfying input values must cause to exercise this path.

Different from the branch conditions along the exercised path, a path condition includes constraints involving only variables from the input arguments, not any local variables

in the program. For example, for the submission in Listing 2, there are two branch conditions in Lines 6 and 11, respectively, and they both involve a local variable (`max` and `min`, respectively). Given an input array `a=[1,2]`, the path condition is `a[1]>a[0] && a[1]>=a[0]`, instead of `a[1]>max && a[1]>=min`.

Note that to cluster submissions, *PaCon* needs their path conditions of the same test. But a structural test generator typically cannot guarantee to produce the same set of tests on each run, and the set of tests generated for any two submissions, respectively, may also be different. For example, for max-difference(A) in Listing 1, *Pex* generates one test as `[18, 18]`, and for max-difference(C) in Listing 3, *Pex* generates two tests: `[32, 48]` and `[40, 41, 18]`. Therefore, it is necessary to use the same set of tests for each of all the submissions so that we can fairly compare the resulting path conditions for each submission over the input space.

In this step, *PaCon* conducts test normalization by taking the union of all the tests produced by the first step for each of all the submissions and placing them into a set where only unique tests are kept. Then *PaCon* invokes the path condition collector on each submission to collect path conditions of the tests in the normalized test set. In this way, the normalized test set for max-difference(A), max-difference(B), and max-difference(C) includes `[18, 18]`, `[32, 48]`, `[40, 41, 18]`, etc.

We leverage *Pex* as the path condition collector. In particular, *Pex* provides a convenient way to attain the path condition of the input value in a test, i.e., by calling the `GetPathConditionString` API method provided by *Pex* . We simply return the result of calling `GetPathConditionString` as the collected path condition.

3.4 Path-Condition-Based Clustering

In this step, *PaCon* conducts path-condition-based clustering to group functionally correct submissions into a set of submission clusters, based on equivalent path-condition-based semantics.

The high-level idea of path-condition-based clustering is that, for two given correct submissions $s1$ and $s2$ for the same problem, *PaCon* checks whether $s1$ and $s2$ have semantically equivalent path conditions of each test in the normalized test set. If so, *PaCon* puts them into the same cluster. It is worth mentioning that the equivalence between submissions is transitive. Therefore, when assessing whether a submission belongs to a given cluster, *PaCon* needs to compare the submission against only one submission randomly chosen from this cluster, instead of against all submissions in it.

The pseudo-code of path-condition-based clustering is shown in Algorithm 1. *PaCon* iterates through all correct submissions (Line 2). For each submission s, *PaCon* traverses the existing clusters to check whether s belongs to any of them by comparing path conditions. Specifically, *PaCon* randomly selects a submission r belonging to an existing cluster

Algorithm 1: path-condition-based clustering

Input: a set of tests T, a set S of functionally correct submissions, and their path conditions PC of each test $\in T$
Output: a set of submission clusters C

```
1  C ← ∅;
2  for submission s ∈ S do
3      foundcluster ← False;
4      for cluster c ∈ C do
5          r ← a submission randomly chosen from c;
6          if isEq(s, r, T, PC) then
7              c ← c ∪ {s};
8              foundcluster ← True;
9              break;
10     if foundcluster == False then
11         c′ ← {s};
12         C ← C ∪ {c′}
13 return C
```

Algorithm 2: $isEq(s_1, s_2, T, PC)$

Input: two functionally correct submissions s_1, s_2, a set of tests T, and their path conditions PC of each test $\in T$
Output: $True$ or $False$

```
1  for test t ∈ T do
2      pc1 ← getPC(PC, s1, t);
3      pc2 ← getPC(PC, s2, t);
4      if not pc1 == pc2 in semantics then
5          return False;
6  return True
```

c as a representative (Line 5), and then calls the function $isEq$ to determine whether s and r are equivalent (Line 6). If $isEq$ returns $True$ (Lines 7-9), cluster c can be regarded as the cluster that the submission s belongs to, and *PaCon* adds s into cluster c and moves on to work on the next submission. If the current cluster set C is empty or there is no cluster found for submission s, *PaCon* builds a new cluster and puts s into it (Lines 10-12). When all submissions are placed in their respective cluster, *PaCon* returns set C as the clusters of submissions.

As shown in Algorithm 2, the equivalence of two submissions s_1 and s_2 is determined by the semantic equivalence of their path conditions. Specifically, for each test $t \in T$, the path conditions of s_1 and s_2 are represented as pc_1 and pc_2, respectively. s_1 and s_2 are considered equivalent, if and only if the semantics of pc_1 and pc_2 are equivalent for each test in T. This equivalence is dedicated to identifying how a program divides the input space into equivalence classes.

Ideally, for two submissions s_1 and s_2 in the same cluster and an arbitrary input t, the equivalence classes of t for $s1$ and $s2$ should be the same. We achieve the semantic comparison of path conditions through a path condition parser and a constraint solver *Z3* [20].

4 Evaluation

In this section, we aim to answer three research questions by empirically investigating the number of clusters produced by *PaCon* and the diversity of syntax in each produced cluster compared to the results produced by syntax-based tools, and exploring what kind of tactics can be reflected by the clusters produced by *PaCon* .

To help answer the first two research questions, besides *PaCon*, we also study two syntax-based clustering tools, which are based on two (near-)duplicate-code detection tools that can work on C# programs: *dupFinder (DF)* [12] from JetBrains, and *Near-Duplicate (ND)* [16] from Microsoft. In particular, to produce clusters from the results of (near-)duplicate-code detection, we put two submissions into the same cluster if there is any duplicate code fragment between them. For each submission that shares no duplicate code fragment with any other submissions, we put this submission into a separate cluster.

4.1 Research Questions

RQ1: What is the number of clusters produced by *PaCon* compared with syntax-based tools? The purpose of RQ1 is to investigate to what extent *PaCon* can potentially reduce the workload of instructors in inspecting programming submissions. One goal of *PaCon* is to relieve instructors from the burden of inspecting an intractable number of submissions. Ideally, with the help of clustering, instructors need to select only one representative from each cluster for inspection, rather than looking into all submissions. However, if the number of clusters produced by a clustering approach remains large, it is still burdensome for instructors to inspect all representatives, and then the clustering approach is not feasible in practice. To answer this question, we count the number of clusters produced by *PaCon*, compared with the number of clusters produced by *ND* and *DF*, respectively.

RQ2: How diverse is the syntax of submissions included in a cluster produced by *PaCon* ? The purpose of RQ2 is to assess the ability of *PaCon* to cluster together submissions with syntactic differences but equivalent path-condition-based semantics. To answer this question, we assess the syntax diversity within each cluster produced by *PaCon*, in short as each *PaCon* cluster. In particular, we measure the syntax diversity within a *PaCon* cluster as the number of syntax-based clusters (i.e., ones produced by a syntax-based tool such as *ND* or *DF*) that the submissions from this *PaCon* cluster belong to. The higher the count, the more syntactically diverse the *PaCon* cluster is.

RQ3: What kind of tactics can be reflected by the clusters produced by *PaCon* ? As mentioned in Section 1, one challenge of identifying tactics comes from the tactic definition that changes with requirements, so it is difficult to label the tactic(s) used in each submission as the ground truth. The purpose of RQ3 is to investigate what kind of tactics *PaCon* can help instructors identify. To answer this question, we manually check each *PaCon* cluster and summarize the tactics used in the submissions for each assignment in the evaluation subjects.

4.2 Evaluation Subjects

The evaluation subjects consist of two sources of real-world data sets.

Code-Hunt. The Code-Hunt data set [2] contains submissions from a 48-hour worldwide coding contest. The contest has four sectors each of which contains six puzzles, and participants (students) were allowed unlimited attempts to solve the programming problems within the time limit. We choose this data set because the difficulty of its puzzles is representative of assignments in introductory-level programming courses and also because the data set is used in related work [3, 4, 18].

Of the 24 total puzzles in the Code-Hunt data set, we first choose the 12 ones whose submissions include at least one branch. Considering that *PaCon* focuses on clustering only functionally correct submissions, we further exclude the submissions that do not perform functionally the same as their given solution in Code-Hunt, keeping only functionally correct ones. In addition, students who participated in the contest were allowed to use either Java or C#. Because our implementation of *PaCon* includes *Pex* (supporting C# without supporting Java) as the test generator, we use a converter [24] to translate Java submissions into C# programs and include only those resulting C# programs that can be successfully compiled. Finally, we exclude puzzles with fewer than ten correct submissions remaining. Table 1 shows the description of the puzzles included in our evaluation. The second column of Table 2 shows the number of submissions included in our evaluation for each of these puzzles.

Sorting. Besides the Code-Hunt data set, our evaluation subjects also include the sorting data set consisting of a number of programs that implement the sorting functionality. We select the sorting problem because (1) sorting is a classical problem taught in CS programming courses, and (2) we want to know how the tactics that *PaCon* can identify are related to those well-labeled problem-solving ways. We collect sorting programs with five different labels: *bubble sort*, *heap sort*, *insertion sort*, *quick sort*, and *merge sort*. We attain the programs for these five kinds of sorting by searching with each label as the keyword among C# programs via Google. The labels of *bubble sort*, *heap sort*, *insertion sort*, *quick sort*, and *merge sort* include 17, 11, 16, 10, and 12 programs, respectively, in total 66 programs.

Table 1. The Description of the Code-Hunt Puzzles Used in Our Evaluation

Puzzle	Description
Sector2-Level2	Count the depth of nesting parentheses in a string
Sector2-Level5	Find the maximum difference between two elements in an array
Sector2-Level6	Generate the string of binary digits for the input integer n
Sector3-Level1	Create a filter that retains only values greater than or equal to a given threshold
Sector3-Level2	Compute the sum of the n-th and $(n-1)$-th Fibonacci numbers
Sector3-Level3	Find the k-th smallest element in an array
Sector3-Level6	Compute the set difference of two input integer arrays
Sector4-Level2	Compute C_m^n, i.e., $\frac{m!}{(n!\times(m-n)!)}$
Sector4-Level6	Advance each character in a string by the Fibonacci number evaluated at the character's integer ASCII value

4.3 RQ1 Results: Number of Produced Clusters

Table 2 shows the statistics of the clustering results. The first two columns indicate the name of an assignment and the total number of functionally correct submissions for this assignment. The number of clusters produced by *PaCon*, *ND*, and *DF* is shown in the last three columns N_c, N_{ND}, and N_{DF}, respectively.

According to the statistics, *PaCon* produces at most 10 clusters for each of the Code-Hunt puzzles and 15 clusters for the sorting problem. The number of clusters produced by *PaCon* is smaller than that of both *ND* and *DF* for all assignments. This gap is more evident as the number of submissions increases. The 99 submissions for Sector3-Level2 are grouped into only 5 clusters by *PaCon*, and the number of clusters produced by *ND* and *DF* is 40 and 68, respectively. In contrast, the 17 submissions for Sector4-Level2 are grouped into 7 clusters by *PaCon*, 10 clusters by *ND*, and 14 clusters by DF.

The trend is consistent with our intuition: for a specific programming problem, the submissions that adopt the same tactic may have distinct ways of syntactically organizing the code, and as the number of submissions increases, this gap tends to increase as well.

In summary, *PaCon* can produce a reasonable number of clusters, and the number of clusters produced by *PaCon* is much smaller than that of the two syntax-based tools (*ND* and *DF*).

4.4 RQ2 Results: Syntax Diversity of Cluster

Figure 2 shows the distribution of *PaCon* clusters' syntax diversity for each assignment. For all the assignments, *PaCon* can produce clusters whose syntax diversity is at least 2. In addition, the clusters with syntax diversity not less than 2 account for at least 40% of the total clusters for most assignments. Moreover, for Sector2-Level2, Sector2-Level5, Sector-2-Level6, Sector3-Level2, etc., *PaCon* even results in clusters whose syntax diversity is greater than 10.

Table 2. The Number of *PaCon* Clusters for the Evaluation Subjects

Assignment	#Submissions	N_c	N_{ND}	N_{DF}
Sector2-Level2	30	3	16	27
Sector2-Level5	99	6	21	61
Sector2-Level6	88	4	28	75
Sector3-Level1	37	6	8	22
Sector3-Level2	99	5	40	68
Sector3-Level3	36	4	7	34
Sector3-Level6	23	10	11	19
Sector4-Level2	17	7	10	14
Sector4-Level6	24	3	15	18
Sorting	66	15	37	55

#Submissions is the number of functionally correct submissions for each assignment. N_c, N_{ND}, and N_{DF} indicate the number of clusters produced by *PaCon*, *ND*, and *DF*, respectively.

When *DF* is used for syntax-based clustering, for all assignments except two (Sector2-Level5 and Sector3-Level1), the percentage of clusters with syntax diversity of at least 2 is not less than 40%. In Sector2-Level6, the percentage is even as high as 75%. In contrast, when *ND* is used for syntax-based clustering, for Sector2-Level5, Sector2-Level6, Sector3-Level1, and Sector3-Level6, the percentage of clusters whose syntax diversity is only 1 exceeds 60%. In particular, for Sector3-Level6, only 10% of *PaCon* clusters have syntax diversity greater than 1.

In summary, *PaCon* can effectively group together submissions with syntactic differences but equivalent path-condition-based semantics.

4.5 RQ3 Results: Tactics Reflected by Clusters

We manually check the clusters produced by *PaCon* and summarize the tactics based on the clusters for each evaluation subject. Due to the space limit, in this subsection,

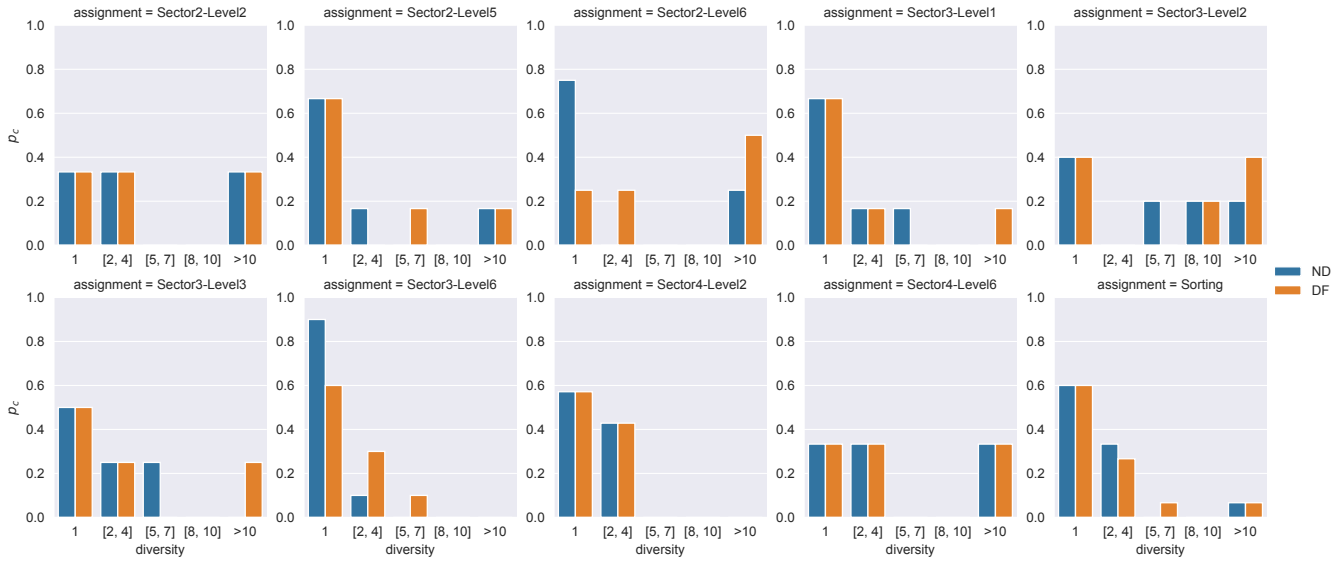

Figure 2. The Distribution of Syntax Diversity in Clusters Produced by *PaCon*

The bars indicate the percentage (p_c) of *PaCon* clusters that contain 1 syntax-based cluster, 2-4 syntax-based clusters, 5-7 syntax-based clusters, etc. The comparison with *DF* is shown in orange bars, and that with *ND* is shown in blue bars.

Table 3. The Description of Clusters and Our Tactic Summary for Sector2-Level5

Tactic Summary	No_C	Clusters	Num
Find the max and min elements of the array, then return their difference	c_0	Call `Max` and `Min` API methods or implement the API methods	66
	c_1	Implement `Max` and `Min` API methods with an extra "=" in the ">=" and "<=" comparison operators, such as "`if (a[i] >= max) {max = a[i];}`"	1
	c_2	Implement `Max` and `Min` API methods with initialization of `max` and `min` using different array elements, such as "`int max = a[0]; int min = a[1];`"	3
Sort first, then return `|a[length-1]-a[0]|`	c_3	Call `Array.sort` API method or implement the API method	26
	c_4	Call `Array.sort` API method and execute some extra code	2
Compute all the pairwise element differences, then return the maximum	c_5	Compute all the pairwise element differences, then return the maximum	1

we discuss the analysis results of only two evaluation subjects. The complete analysis results can be found at https://sites.google.com/view/paconproj/.

4.5.1 Case of Sector2-Level5. Table 3 shows the detailed results of Sector2-Level5. We summarize three tactics based on the clustering results of *PaCon*: "*find the max and min elements of the array, then return their difference*", "*sort first, then return* `|a[length-1]-a[0]|`", and "*compute all the pairwise element differences, then return the maximum*".

For the first tactic in the table, *PaCon* produces three clusters among which most of the submissions fall into cluster c_0. Submissions in cluster c_0 either call the built-in `Max` and `Min` API methods or implement the two API methods themselves. There are also a few other submissions that adopt similar tactics as those in cluster c_0 but are not grouped into cluster

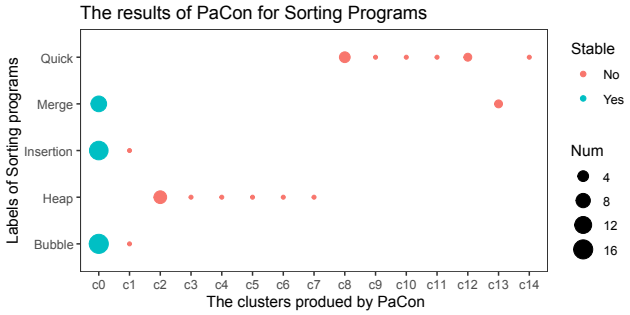

Figure 3. The Results of *PaCon* on Sorting Programs Compared with Their Labels

c_0. For example, cluster c_1 includes one submission that implements the `Max` and `Min` API methods with extra "=" in the

">=" and "<=" comparison operators. To get the maximum element in array a, most submissions use statements such as "`if (a[i]>max) {max = a[i];}`", whereas the submission in cluster c_1 uses the statement "`if(a[i]>=max) {max = a[i];}`". In addition, cluster c_2 includes three submissions that initialize the `max` and `min` variables as different array elements from those used in cluster c_0 or c_1.

Although the operator difference with extra "=" seems to be of only syntactic nature, in some cases it does lead to semantic differences of path conditions. Due to extra "=", not only the operator changes, but the comparisons also occur for different pairs of elements. For example, consider that we are calculating the maximum element in an array `a = [2, 2, 1, 3]`. When we use "`if(a[i]>max)`" for the update of `max`, the path condition is `a[1]<=a[0] && a[2]<=a[0] && a[3]>a[0]`. But when we use "`if(a[i]>=max)`" for the update, the path condition becomes `a[1]>=a[0] && a[2]<a[1] && a[3]>=a[1]`.

For the second tactic in the table, *PaCon* produces two clusters c_3 and c_4, and most of the submissions fall into cluster c_3. Submissions in clusters c_3 first conduct sorting of the given array elements, whereas besides conducting the sorting, those in c_4 also execute some extra code without which the submissions still have the same return values but whose execution introduces additional constraints in the path conditions. For the third tactic in the table, there is one cluster with only one submission that computes all the pairwise element differences and then returns the maximum.

4.5.2 Case of the Sorting Problem.

Figure 3 shows a scatter plot for the clustering results produced by *PaCon* on the sorting programs. The x axis indicates the clusters produced by *PaCon*, and the y axis indicates five different labels for the sorting programs. The number of programs is reflected by the size of scatters.

According to the results shown in Figure 3, instead of identifying different sorting labels, the clustering results produced by *PaCon* reflect the stability of sorting. A sorting program is stable if any two elements with equal values in an array always appear in the same order in the program's output as they appear in the input [28]. Specifically, the programs in $c0$ are all stable sorting programs, consisting of most bubble sort programs, most insertion sort programs, and most merge sort programs. The programs in the other 14 clusters are all unstable sorting programs.

Cluster $c1$ consists of one program of bubble sort and one program of insertion sort. For the program implementing bubble sort in $c1$, instead of comparing adjacent elements, in each round, the program compares a fixed element with each element following it and swaps the two elements when an inversion is found. It is not a stable sorting program. The program of insertion sort in cluster $c1$ actually implements selection sort, and is also unstable.

Cluster $c13$ consists of two programs implementing merge sort. When merging two ordered subarrays, the programs put `a[i]` to the left of `a[j]` only when `a[i] < a[j]` (`a[i]` is an element in the left subarray and `a[j]` is an element in the right subarray). This operation causes the relative order of the two elements `a[i]` and `a[j]` to be reversed if their values are equal, so these two programs are not stable sorting programs.

Programs in each of the remaining 12 clusters are all with only one label. These programs implement either heap sort or quick sort, and they are all unstable.

The remaining question is why *PaCon* divides those unstable sorting programs into different clusters. The key reason is that the set of array input values for exposing the instability of an unstable sorting program can be different from the set for exposing the instability of another unstable sorting program[1]. For instance, consider two quick sort programs Q_1 and Q_2 that use different criteria to select the pivot `p`, and an input array `arr = [1, 2a, 2b, 3, 4]`. Suppose that program Q_1 takes the leftmost element as the pivot `p`, and program Q_2 takes the rightmost element as the pivot `p`. Both Q_1 and Q_2 put elements smaller than `p` to its left and put elements not smaller than `p` to its right. After sorting, the output of Q_1 is still `[1, 2a, 2b, 3, 4]`, but the output of Q_2 becomes `[1, 2b, 2a, 3, 4]`. Although Q_1 and Q_2 are both unstable, the instability of Q_2 is exposed by the input `arr`, but that of Q_1 is not. Recall that *PaCon* requires that programs in the same cluster must have equivalent path conditions for each generated test, so *PaCon* divides unstable sorting programs into different clusters.

To further confirm our finding, after we add a precondition of "*all elements are distinct*" to the input of sorting, *PaCon* groups all sorting algorithms into the same cluster.

4.6 Summary of Results

In our evaluation, the number of clusters produced by *PaCon* is reasonable and much less than that produced by syntax-based tools. At the same time, the clusters effectively groups together those submissions with high syntax diversity while sharing equivalent path-condition-based semantics. Based on our manual analysis, the clusters produced by *PaCon* can be promising to characterize how the submissions differ in their ways to solve the target problem. For the sorting problem, although *PaCon* does not help recognize the common sorting algorithms, it can still distinguish a significant characteristic of sorting programs: whether a sorting program is stable or not.

5 Discussion

There are two major limitations of *PaCon* in terms of its current implementation and design.

[1]Note that even an unstable sorting program does not always swap two equal elements in the given array input value.

First, the implementation of *PaCon* includes a constraint solver, such as *Z3* [20], to check the equivalence of path conditions, but there exist some cases (such as constraints about `HashSet`) that *Z3* cannot deal with. If such a situation happens, *PaCon* directly regards the two path conditions as nonequivalent (if their string forms are not the same) and may divide submissions using the same tactic into separate clusters. In our evaluation, *PaCon* encounters this situation when clustering submissions for Sector3-Level6.

Second, *PaCon* uses the semantic equivalence of path conditions to cluster submissions. It pays more attention to the semantic of programs and can be robust in the face of many syntactic differences. However, this trade-off has limitations as well as benefits. When two submissions have two different tactics, and the tactic difference manifests in some program features (such as the syntactic composition of constraints within a path condition) yet without impacting the semantics of the path conditions, *PaCon* puts these two submissions into the same cluster. In our evaluation, *PaCon* encounters this situation when clustering submissions for Sector4-Level2.

6 Related Work

Due to the growing nature of CS courses, clustering of programming submissions becomes an appealing approach to help quickly inspect a large number of submissions. A substantial body of work has been proposed for this task. In this section, we discuss a number of closely related approaches.

CLARA [9], *MistakeBrowser* [10], and *TipsC* [22] all aim to provide feedback to incorrect submissions. *CLARA* [9] and *TipsC* [22] both cluster correct programming submissions. In particular, *CLARA* puts two submissions into the same cluster if they have the same control flow structure and there exists a bijective relation between their variables. *TipsC* [22] first normalizes the programs to linear representations and then clusters "similar" programs according to a variant of the Levenshtein edit distance. Different from *CLARA* and *TipsC*, *MistakeBrowser* [10] first learns code transformations (i.e., code edits) to correct incorrect submissions, and then uses the learned transformations to cluster incorrect submissions.

OverCode [6] is a system for visualizing and exploring the variations in programming submissions. When clustering submissions, *OverCode* first produces the cleaned code of programs by renaming common variables that have identical sequences across two or more program traces, and then groups the submissions whose cleaned code contains identical sets of program statements.

PaCon differs from *CLARA*, *MistakeBrowser*, *TipsC*, and *OverCode* mainly in terms of the design goals. *PaCon* aims to help instructors identify tactics in functionally correct submissions, whereas the proceeding clustering approaches aim to generate feedback to incorrect submissions or to visualize the variations in submissions. Due to such differences, *PaCon* pays more attention to the semantic features of programs, whereas *CLARA*, *MistakeBrowser*, *TipsC*, and *OverCode* mainly take syntactic features into account.

SolMiner [15] leverages static program analysis, data mining, and machine learning to mine distinct solutions (with different data structures, space-time complexity, etc.) from a large pool of submissions. Given that *SolMiner* represents a program as a sequence of mini-ASTs, each of which corresponds to a portion of a basic block in the program, *SolMiner* is less robust to syntactic differences in submissions than *PaCon*.

SemCluster [18] clusters submissions (no matter correct or incorrect) based on their vector representation. The vector consists of two quantitative semantic features: the control flow feature and the data flow feature. Given a program and a test suite, the control flow feature tracks the number of input values flowing through the same control flow paths as different tests, whereas the data flow feature tracks the number of times that a specific value in memory is changed to another specific value. To enable the computation of the control flow feature (involving model counting), *SemCluster* requires instructors to provide bounds on the input values, whereas *PaCon* has no such requirement. In addition, *SemCluster* leverages a classical clustering algorithm based on *similarity* of quantitative semantic features across submissions whereas *PaCon* conducts clustering with the help of the *Z3* constraint solver [20] based on the *equivalence* of path condition semantics across submissions.

Grasa [17] aims to augment a given test suite with a minimal set of generated tests whose purpose is to detect a maximum number of incorrect submissions. Different from *PaCon* , to accomplish this objective, *Grasa* clusters incorrect submissions by approximating their behavioral equivalence to each other.

7 Conclusion

In this paper, we have raised the awareness of identifying different problem-solving ways (named as *tactics*) in programming submissions and proposed a symbolic analysis approach named *PaCon* for clustering functionally correct programming submissions to provide a way of identifying tactics. Different from existing syntax-based approaches, *PaCon* determines the semantic equivalence of programs by the semantic equivalence of their path conditions. Our evaluation results on real-world data sets show that *PaCon* can produce a reasonable number of clusters each of which effectively groups together those submissions with high syntax diversity while sharing equivalent path-condition-based semantics, providing a promising way toward identifying tactics.

References

[1] S. Bhatia, P. Kohli, and R. Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proc. 40th IEEE/ACM International Conference on Software Engineering*. 60–70. https://doi.org/10.1145/3180155.3180219

[2] Judith Bishop, R. Nigel Horspool, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2015. Code Hunt: Experience with Coding Contests at Scale. In *Proc. 37th IEEE/ACM International Conference on Software Engineering*. 398–407. https://doi.org/10.1109/ICSE.2015.172

[3] S. Combéfis and A. Schils. 2016. Automatic Programming Error Class Identification with Code Plagiarism-Based Clustering. In *Proc. 2nd International Code Hunt Workshop on Educational Software Engineering*. 1–6. https://doi.org/10.1145/2993270.2993271

[4] L. D'Antoni, R. Samanta, and R. Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Proc. 28th International Conference on Computer Aided Verification*. 383–401. https://doi.org/10.1007/978-3-319-41540-6_21

[5] A. Gerdes, B. Heeren, and J. Jeuring. 2012. Teachers and Students in Charge. In *Proc. 21st Century Learning for 21st Century Skills - 7th European Conference of Technology Enhanced Learning*. 383–388. https://doi.org/10.1007/978-3-642-33263-0_31

[6] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput. Hum. Interact.* 22, 2 (2015), 1–35. https://doi.org/10.1145/2699751

[7] S. Gulwani. 2014. Example-Based Learning in Computer-aided STEM Education. *Commun. ACM* 57, 8 (2014), 70–80. https://doi.org/10.1145/2634273

[8] S. Gulwani, I. Radiček, and F. Zuleger. 2014. Feedback Generation for Performance Problems in Introductory Programming Assignments. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 41–51. https://doi.org/10.1145/2635868.2635912

[9] S. Gulwani, I. Radiček, and F. Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 465–480. https://doi.org/10.1145/3192366.3192387

[10] A. Head, E. L. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proc. 4th ACM Conference on Learning @ Scale*. 89–98. https://doi.org/10.1145/3051457.3051467

[11] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *Proc. 34th IEEE/ACM International Conference on Automated Software Engineering*. 388–398. https://doi.org/10.1109/ASE.2019.00044

[12] JetBrains. 2021. DupFinder Command-line Tool. https://www.jetbrains.com/help/resharper/dupFinder.html.

[13] H. Keuning, B. Heeren, and J. Jeuring. 2014. Strategy-Based Feedback in a Programming Tutor. In *Proc. Computer Science Education Research Conference*. 43–45. https://doi.org/10.1145/2691352.2691356

[14] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[15] L. Luo and Q. Zeng. 2016. SolMiner: Mining Distinct Solutions in Programs. In *Proc. 38th IEEE/ACM International Conference on Software Engineering Companion*. 481–490. https://doi.org/10.1145/2889160.2889202

[16] Microsoft. 2021. Near-Duplicate Code Detector. https://github.com/microsoft/near-duplicate-code-detector.

[17] J. Osei-Owusu, A. Astorga, L. Butler, T. Xie, and G. Challen. 2019. Grading-Based Test Suite Augmentation. In *Proc. 34th IEEE/ACM International Conference on Automated Software Engineering*. 226–229. https://doi.org/10.1109/ASE.2019.00030

[18] D. Perry, D. Kim, R. Samanta, and X. Zhang. 2019. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 860–873. https://doi.org/10.1145/3314221.3314629

[19] Online Course Report. 2017. The 50 Most Popular MOOCs of All Time. https://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/.

[20] Microsoft Research. 2021. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[21] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proc. 39th IEEE/ACM International Conference on Software Engineering*. 404–415. https://doi.org/10.1109/ICSE.2017.44

[22] S. Sharma, P. Agarwal, P. Mor, and A. Karkare. 2018. TipsC: Tips and Corrections for Programming MOOCs. In *Proc. 19th International Conference on Artificial Intelligence in Education*. 322–326. https://doi.org/10.1007/978-3-319-93846-2_60

[23] R. Singh, S. Gulwani, and A. Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 15–26. https://doi.org/10.1145/2491956.2462195

[24] Tangible Software Solutions. 2004. C++ to C# Converter. https://www.tangiblesoftwaresolutions.com/product_details/cplusplus_to_csharp_converter_details.html

[25] N. Tillmann and J. de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Proc. 2nd International Conference of Tests and Proofs*. 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[26] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized Unit Tests. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 253–262. https://doi.org/10.1145/1081706.1081749

[27] K. Wang, R. Singh, and Z. Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 481–495. https://doi.org/10.1145/3192366.3192384

[28] Wikipedia. 2020. Category: Stable sorts. https://en.wikipedia.org/wiki/Category:Stable_sorts

[29] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proc. 11th Joint Meeting on Foundations of Software Engineering*. 740–751. https://doi.org/10.1145/3106237.3106262