Feedback Generation for Performance Problems in Introductory Programming Assignments *

Sumit Gulwani
Microsoft Research, USA
sumitg@microsoft.com

Ivan Radiček
TU Wien, Austria
radicek@forsyte.at

Florian Zuleger
TU Wien, Austria
zuleger@forsyte.at

ABSTRACT

Providing feedback on programming assignments manually is a tedious, error prone, and time-consuming task. In this paper, we motivate and address the problem of generating feedback on performance aspects in introductory programming assignments. We studied a large number of functionally correct student solutions to introductory programming assignments and observed: (1) There are different algorithmic strategies, with varying levels of efficiency, for solving a given problem. These different strategies merit different feedback. (2) The same algorithmic strategy can be implemented in countless different ways, which are not relevant for reporting feedback on the student program.

We propose a light-weight programming language extension that allows a teacher to define an algorithmic strategy by specifying certain key values that should occur during the execution of an implementation. We describe a dynamic analysis based approach to test whether a student's program matches a teacher's specification. Our experimental results illustrate the effectiveness of both our specification language and our dynamic analysis. On one of our benchmarks consisting of 2316 functionally correct implementations to 3 programming problems, we identified 16 strategies that we were able to describe using our specification language (in 95 minutes after inspecting 66, i.e., around 3%, implementations). Our dynamic analysis correctly matched each implementation with its corresponding specification, thereby automatically producing the intended feedback.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging; I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming—*Automatic analysis of algorithms*

*The second and third author were supported by the Vienna Science and Technology Fund (WWTF) grant ICT12-059.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the author/owner(s). Publication rights licensed to ACM.

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635912

General Terms

Algorithms, Languages, Performance.

Keywords

Education, MOOCs, performance analysis, trace specification, dynamic analysis.

1. INTRODUCTION

Providing feedback on programming assignments is a very tedious, error-prone, and time-consuming task for a human teacher, even in a standard classroom setting. With the rise of Massive Open Online Courses (MOOCs) [16], which have a much larger number of students, this challenge is even more pressing. Hence, there is a need to introduce automation around this task. Immediate feedback generation through automation can also enable new pedagogical benefits such as allowing resubmission opportunity to students who submit imperfect solutions and providing immediate diagnosis on class performance to a teacher who can then adapt her instruction accordingly [9].

Recent research around automation of feedback generation for programming problems has focused on guiding students to functionally correct programs either by providing counterexamples [26] (generated using test input generation tools) or generating repairs [22]. However, non-functional aspects of a program, especially performance, are also important. We studied several programming sessions of students who submitted solutions to introductory C# programming problems on the Pex4Fun [4] platform. In such a programming session, a student submits a solution to a specified programming problem and receives a counterexample based feedback upon submitting a functionally incorrect attempt (generated using Pex [25]). The student may then inspect the counterexample and submit a revised attempt. This process is repeated until the student submits a functionally correct attempt or gives up. We studied 24 different problems, and observed that of the 3993 different programming sessions, 3048 led to functionally correct solutions. However, unfortunately, on average around 60% of these functionally correct solutions had (different kinds of) performance problems. In this paper, we present a methodology for semi-automatically generating appropriate performance related feedback for such functionally correct solutions.

From our study, we made two observations that form the basis of our semi-automatic feedback generation methodology. (i) There are different *algorithmic strategies* with varying levels of efficiency, for solving a given problem. Algorithmic

strategies capture the global high-level insight of a solution to a programming problem, while also defining key performance characteristics of the solution. Different strategies merit different feedback. (ii) The same algorithmic strategy can be implemented in countless different ways. These differences originate from local low-level implementation choices and are not relevant for reporting feedback on the student program.

In order to provide meaningful feedback to a student it is important to identify what algorithmic strategy was employed by the student program. A profiling based approach that measures running time of a program or use of static bound analysis techniques [10, 12] is not sufficient for our purpose, because different algorithmic strategies that necessitate different feedback may have the same computational complexity. Also, a simple pattern matching based approach is not sufficient because the same algorithmic strategy can have syntactically different implementations.

Our key insight is that the algorithmic strategy employed by a program can be identified by observing the values computed during the execution of the program. We allow the teacher to specify an algorithmic strategy by simply annotating (at the source code level) certain key values computed by a sample program (that implements the corresponding algorithm strategy) using a new language construct, called *observe*. Fortunately, the number of different algorithmic strategies for introductory programming problems is often small (at most 7 per problem in our experiments). These can be easily enumerated by the teacher in an iterative process by examining any student program that does not match any existing algorithmic strategy; we refer to each such step in this iterative process as an *inspection* step.

We propose a novel dynamic analysis that decides whether the student program (also referred to as an *implementation*) matches an algorithm strategy specified by the teacher in the form of an annotated program (also referred to as a *specification*). Our dynamic analysis executes a student's implementation and the teacher's specification to check whether the key values computed by the specification also occur in the corresponding traces generated from the implementation.

We have implemented the proposed framework in C# and evaluated our approach on 3 pre-existing programming problems on Pex4Fun (attempted by several hundreds of students) and on 21 new problems that we hosted on Pex4Fun as part of a programming course (attempted by 47 students in the course). Experimental results show that: (i) The manual teacher effort required to specify various algorithmic strategies is a small fraction of the overall task that our system automates. In particular, on our MOOC style benchmark of 2316 functionally correct implementations to 3 pre-existing programming problems, we specified 16 strategies in 95 minutes after inspecting 66 implementations. On our standard classroom style benchmark of 732 functionally correct implementations to 21 programming problems, we specified 66 strategies in 266 minutes after inspecting 149 implementations. (ii) Our methodology for specifying and matching algorithmic strategies is both expressive and precise. In particular, we were able to specify all 82 strategies using our specification language and our dynamic analysis correctly matched each implementation with the intended strategy.

This paper makes the following contributions:

- We observe that there are different algorithmic strategies used in functionally correct attempts to introductory programming assignments; these strategies merit different performance related feedback.
- We describe a new language construct, called *observe*, for specifying an algorithmic strategy (§3).
- We describe a dynamic analysis based approach to test whether a student's implementation matches the teacher's specification (§4).
- Our experimental results illustrate the effectiveness of our specification language and dynamic analysis (§6).

2. OVERVIEW

In this section we motivate our problem definition and various aspects of our solution by means of various examples.

2.1 Motivation

Fig. 1 shows our running examples. Programs (a)-(i) (**IM**) show some sample implementations for the *anagram problem* (which involves testing whether the two input strings can be permuted to become equal) on the Pex4Fun platform. All 9 programs are examples of inefficient implementations, because of their *quadratic asymptotic complexity*. An efficient solution, for example, is to first collect (e.g. in an array) the number of occurrences of each character in both strings and then compare them, leading to *linear asymptotic complexity*.

Algorithmic strategies. In implementations **IM** we identify *three different algorithmic strategies*. Implementations **C1**-**C3** iterate over one of the input strings and for each character in that string count the occurrences of that character in both strings (*counting strategy*). Implementations **S1**-**S3** sort both input strings and check if they are equal (*sorting strategy*). Implementations **R1**-**R3** iterate over one of the input strings and remove corresponding characters from the other string (*removing strategy*).

Implementation details. An algorithmic strategy can have several implementations. In case of counting strategy: Implementation **C1** calls manually implemented method *countChar* to count the number of characters in a string (lines 5 and 6), while implementation **C2** uses a special C# construct (lines 6 and 7) and implementation **C3** uses the library function *Split* for that task (lines 4 and 5). In case of the sorting strategy: Implementation **S1** employs binary insertion sort, while implementation **S2** employs bubble sort and implementation **S3** uses a library call (lines 4 and 5). We also observe different ways of removing a character from a string in implementations **R1**-**R3**.

Desired feedback. Each of the three identified strategies requires separate feedback (independent of the underlying implementation details), to help a student understand and fix the performance issues. For the first strategy (implementations **C1**-**C3**), a possible feedback might be: *"Calculate the number of characters in each string in a preprocessing phase, instead of each iteration of the main loop"*; for the second strategy (**S1**-**S3**), it might be: *"Instead of sorting input strings, compare the number of character occurrences in each string"*; and for the third strategy (**R1**-**R3**): *"Use a more efficient data-structure to remove characters"*.

2.2 Specifying Algorithmic Strategies

Key values. Our key insight is that different implementations that employ the same algorithmic strategy generate the same *key values* during their execution on the same input. For example, (the underlined expressions in) the implementations **C1** and **C2** both produce the key value sequence

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length) return false;

  foreach (Char ch in s.ToCharArray()){
    if (countChars(s, ch)
          != countChars(t, ch)){
      return false;
    }}
  return true;}

int countChars(String s, Char c){
  int number = 0;

  foreach (Char ch in s.ToCharArray()){
    if (ch == c){
      number++;
    }}
  return number;}
```

(a) Counting/Manual (**C1**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  else
    return s.All(c =>
      s.Where(c2 => c2 == c).Count() ==
      t.Where(c2 => c2 == c).Count()
      );
}
```

(b) Counting/Library (**C2**)

```
bool Puzzle(string s, string t) {
  if(s.Length != t.Length) return false;
  foreach (var item in s) {
    if(s.Split(item).Length
        != t.Split(item).Length)
      return false;
  }
  return true; }
```

(c) Counting/Split (**C3**)

```
int BinarySearch(List<char> xs, char y) {
  int low = 0, high = xs.Count;
  while (low < high) {
    int mid = (high - low) / 2 + low;
    if (y < xs[mid]) high = mid;
    else if (y > xs[mid]) low = mid + 1;
    else return mid;}
  return low;}

char[] Sort(string xs) {
  var res = new List<char>();
  foreach (var x in xs) {
    var pos = BinarySearch(res, x);
    res.Insert(pos, x);}
  return res.ToArray(); }

bool Puzzle(string s, string t) {
  return String.Join("", Sort(s))
    == String.Join("", Sort(t)); }
```

(d) Sorting/Binary Insertion (**S1**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length) return false;
  char[] sa = s.ToCharArray();
  char[] ta = t.ToCharArray();
  for (int j=0; j < sa.Length; j++) {
    for (int i=0; i<sa.Length - 1;i++) {
      if (sa[i]<sa[i+1]){ char temp=sa[i];
        sa[i]=sa[i+1]; sa[i+1]=temp;}
      if (ta[i]<ta[i+1]){ char temp=ta[i];
        ta[i] = ta[i+1]; ta[i+1] = temp;}
    }}
  for (int k = 0; k < sa.Length; k++) {
    if (sa[k] != ta[k]) return false; }
  return true; }
```

(e) Sorting/Bubble (**S2**)

```
bool Puzzle(string s, string t) {
  var sa = s.ToCharArray();
  var ta = t.ToCharArray();
  Array.Sort(sa);
  Array.Sort(ta);
  return sa.SequenceEqual(ta);}
```

(f) Sorting/Library (**S3**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length) return false;
  foreach (char c in t.ToCharArray()) {
    int index = s.IndexOf(c);
    if (index < 0) return false;
    s = s.Remove(index, 1); }
  return true; }
```

(g) Removing/Library (**R1**)

```
bool Puzzle(string s, string t) {
  return IsPermutation(s, t);
}
bool IsPermutation(String s, string t) {
  if (s == t) return true;
  if (s.Length != t.Length) return false;
  int index = t.IndexOf(s[0]);
  if (index == -1) return false;

  s = s.Substring(1);
  t = t.Remove(index, 1);

  return IsPermutation(s, t);
}
```

(h) Removing/Recursive (**R2**)

```
bool Puzzle(string s, string t) {
  char[] sc = s.ToCharArray();
  char[] tc = t.ToCharArray();
  Char c = '#';
  if(sc.Length!=tc.Length) return false;
  for(int i=0;i<sc.Length;i++) {
    c = sc[i];
    for(int j=0;j<tc.Length;j++) {
      if(tc[j]==c){
        tc[j]='#';
        break;}
      if(j==tc.Length-1) {
        return false; }}}
  return true; }
```

(i) Removing/Manual (**R3**)

```
Puzzle(string s, string t) {
  if (nd1) { string tt = t; t = s; s = tt; }
  for (int i = 0; i < s.Length; ++i) {
    int cnt1 = 0, cnt2 = 0;
    for (int j = 0; j < s.Length; ++j) {
      if (s[j] == s[i]) {
        if (nd2) observe(s[j]);
        cnt1++;
      }}
    if (!nd2) observeFun(Split());
    observe(nd2 ? cnt1 : cnt1 + 1);
    for (int j = 0; j < t.Length; ++j) {
      if (t[j] == s[i]) {
        if (nd2) observe(t[j]);
        cnt2++;
      }}
    if (!nd2) observeFun(Split());
    observe(nd2 ? cnt2 : cnt2 + 1); }}
```

(j) Counting Specification (**CS**)

```
Puzzle(string s, string t) {
  if (nd1) s = s.ToUpperInvariant();
  char[] ca = s.ToCharArray();
  Array.Sort(ca);
  if (nd2) Array.Reverse(ca);
  observe(ca);
}
```

(k) Sorting Specification (**SS**)

```
Puzzle(string s, string t) {
  if (nd1) {string tt = t; t = s; s = tt;}
  for (int i = 0; i < s.Length; ++i) {
    if (s.Substring(i) == t) return;
    int ni = nd2 ? i : s.Length - i - 1;
    int k = nd3 ? t.IndexOf(s[ni])
                : t.LastIndexOf(s[ni]);
    t = t.Remove(k, 1);
    observe(t, CompareLetterString); }}
```

(l) Removing Specification (**RS**)

```
bool Puzzle(string s, string t) {
  if(s.Length != t.Length) return false;
  Char[] taux = t.ToCharArray();
  for(int i = 0; i < s.Length; i++) {
    Char sc = s[i];
    Boolean exists = false;
    for(int j = 0; j < t.Length; j++) {
      if(sc == taux[j]) {
        exists = true; taux[j] = ' ';
        break; }}
    if(exists == false) return false; }
  return true; }
```

(m) Removing/Manual 2 (**R4**)

```
bool CompareLetterString(string a,string b){
  var la = a.Where(x=>char.IsLetter(x));
  var lb = b.Where(x=>char.IsLetter(x));
  return la.SequenceEqual(lb);
}
```

(n) Custom Data Equality (**CDE**)

```
bool Puzzle(string s, string t) {
  if (s.Length !=t.Length) return false;
  int [] cs=new int [256];
  int [] ct=new int [256];
  for(int i=0;i<s.Length;i++)
    cs[(int) s[i]]++;
  for(int i=0;i<t.Length;++i)
    ct[(int) t[i]]++;
  for (int i=0;i<256;i++)
    if(cs[i] != ct[i]) return false;
  return true;
}
```

(o) Efficient/Compare (**E1**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  char[] cs = s.ToCharArray();
  char[] ct = t.ToCharArray();
  int[] hash = new int[256];
  for (int i=0; i<255; ++i) {
    hash[i] = 0;
  }
  foreach (char ch in cs) {
    hash[(int)ch]++; }
  foreach (char ch in ct) {
    hash[(int)ch]--; }
  for (int i=0; i<255; ++i) {
    if (hash[i] < 0)
      return false; }
  return true; }
```

(p) Efficient/Difference (**E2**)

```
void Puzzle(string s, string t) {
  if (nd1){string tt = t; t = s; s = tt;}
  int[] cs = new int[256],ct = new int[256];
  cover(ToCharArray());
  cover(ToCharArray());
  cover(255);
  for (int i = 0; i < s.Length; ++i) {
    cs[(int)s[i]]++;
    observe(cs); }
  for (int i = 0; i < t.Length; ++i) {
    if (nd2) { cs[(int)t[i]]--;
      observe(cs);
    } else { ct[(int)t[i]]++;
      observe(ct);
    }
  }
  cover(255); }
```

(q) Efficient Specification (**ES**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length) return false;
  string cp = t;
  for(int i=0; i<s.Length; i++) {
    char k = s[i]; bool found = false;
    for(int j=0; j<cp.Length; j++) {
      if (cp[j] == k) {
        if (j == 0) {
          cp = (Char)0+cp.Substring(1);}
        else if(j == cp.Length - 1) {
          cp = cp.Substring(0, j)+(Char)0;}
        else {
          cp = cp.Substring(0, j) +
            (Char) 0 + cp.Substring(j + 1);}
        found = true; break; }}
    if (!found) return false; }
  return true; }
```

(r) Removing/Separate computation (**R5**)

Figure 1: Running example: Implementations and Specifications of Anagram assignment.

$(a, b, a, 2, b, a, a, 2, a, b, a, 1, b, a, a, 1, a, b, a, 2, b, a, a, 2)$ on the input strings "aba" and "baa".

Our framework allows a teacher to describe an algorithmic strategy by simply annotating certain expressions in a sample implementation using a special language statement `observe`. Our framework decides whether or not a student implementation matches a teacher specification by comparing their execution traces on common input(s). We say that an implementation Q *matches* a specification P, if (1) the execution trace of P is a subsequence of the execution trace of Q, and (2) for every observed expression in P there is an expression in Q that has generated the same values. We call this matching criterion a *trace embedding*. The notion of trace embedding establishes a fairly strong connection between specification and implementation: basically, both programs produce the same values at corresponding locations in the same order. Our notion of trace embedding is an adaptation of the notion of a simulation relation [17] to dynamic analysis.

Non-deterministic choice. Because of minor differences between implementations of the same strategy, key-values can differ. For example, implementation **C3** uses a library function to obtain the number of characters, while implementations **C1** and **C2** explicitly count them by explicitly iterating over the string. Moreover, counted values in **C3** are incremented by one compared to those in **C1** and **C2**. **C3** thus yields a different, but related, trace (Split, 3, Split, 3, Split, 2, Split, 2, Split, 3, Split, 3) on input strings "aba" and "baa". To address variations in implementation details, we include a *non-deterministic* choice construct in our specification language. The non-determinism is fixed before the execution; thus such a choice is merely a syntactic sugar to succinctly represent *multiple similar specifications* (n non-deterministic variables $= 2^n$ specifications).

Specifications. **CS**, **SS**, and **RS** denote the specifications for the counting strategy (used in implementations **C1**-**C3**), sorting strategy (used in **S1**-**S3**), and removing strategy (used in **R1**-**R3**) respectively. In **CS**, the teacher observes the characters that are iterated over (lines 7 and 14), the results of counting the characters (lines 11 and 18), and use of library function *Split* (lines 10 and 17). Also the teacher uses *non-deterministic* Boolean variables: *nd1* (line 2) to choose the string over which the main loop iterates (as the input strings are symmetric in the anagram problem); and *nd2* to choose between manual and library function implementations (which also decides on observed counted values on lines 18 and 11). In **SS** the teacher observes one of the input strings after sorting, and non-deterministically allows that implementations convert input string to upper-case (*nd1* on line 2), and sort the string in reverse order (*nd2* on line 5). Notice that it is enough to observe only one sorted input, as in the case that the input strings are anagrams, the sorted strings are the same. In **RS** the teacher observes the string with removed characters and non-determinstcally chooses which string is iterated (*nd1* on line 2), direction of the iteration (*nd2* on line 5) and the direction in which the remove candidate is searched for (*nd3* on line 6).

3. SPECIFICATIONS AND IMPLEMENTATIONS

In this section we introduce an imperative programming language $\mathcal{L}$ that supports standard constructs for writing implementations, and has some novel constructs for writing specifications.

Expression e	::=	$d \mid v \mid v_1\ op_{bin}\ v_2 \mid op_{un}\ v \mid v_1[v_2]$
Statement s	::=	$v := e \mid v_1[v_2] := e \mid v := \texttt{f}(v_1, \ldots, v_n)$
	|	$s_0; s_1 \mid \texttt{while}\ v\ \texttt{do}\ s \mid \texttt{skip}$
	|	$\texttt{if}\ v\ \texttt{then}\ s_0\ \texttt{else}\ s_1$
	|	$\texttt{observe}(v, [E])$
	|	$\texttt{observeFun}(\texttt{f}[v_1, \ldots, v_n], [E])$

Figure 2: The syntax of $\mathcal{L}$ language.

3.1 The Language $\mathcal{L}$

The syntax of the language $\mathcal{L}$ is stated in Fig. 2. We discuss the features of the language below.

Expressions. *A data value* d is any value from some *data domain* set D, which contains all values in the language (e.g., in C#, all integers, characters, arrays, hashsets, ...). A *variable* v belongs to a (finite) set of variables *Var*. An *expression* is either a data value d, a variable v, an operator applied to variables v_1, v_2 or an array access $v_1[v_2]$. Here, op_{bin} represents a set of binary operators (e.g., $+, \cdot, =, <, \wedge$) and op_{un} a set of unary operators (e.g., $\neg, |\cdot|$). We point out that the syntax of $\mathcal{L}$ ensures that programs are in *three address code*: operators can only be applied to variables, but not to arbitrary expressions. The motivation for this choice is that three address code enables us to observe any expression in the program by observing only variables. We point out that any program can be (automatically) translated into three address code by assigning each subexpression to a new variable. For example, the statement $v_1 := v_2 + (a + b)$ can be translated into three-address code as follows: $v_3 := a + b; v_1 := v_2 + v_3$. This enables us to observe the subexpression $a + b$ by observing v_3.

Statements. The statements of $\mathcal{L}$ allow to build simple imperative programs: assignments (to variables and array elements), `skip` statement, composition of statements, looping and branching constructs. We also allow library function calls in $\mathcal{L}$, denoted by $v := \texttt{f}(v_1, \ldots, v_n)$, where $\texttt{f} \in F$ is a library function name, from a set of all library functions F. There are two special `observe` constructs, which are only available to the teacher (and not to the student). We discuss the observe statements in §3.3 below. We assume that each statement s is associated with a *unique program location* ℓ, and write $\ell : s$.

Functions. For space reasons we do not define functions here. We could easily extend the language to (recursive) functions. In fact we allow (recursive) functions in our implementation.

Semantics. We assume some standard imperative semantics to execute programs written in the language $\mathcal{L}$ (e.g., for C# we assume the usual semantics of C#). The two `observe` statements have the same semantic meaning of the `skip` statement.

Computation domain. We extend the data domain D by a special symbol ?, which we will use to represent *any* data value. We define the *computation domain Val* associated with our language $\mathcal{L}$ as $Val = D \cup (F \times D^*)$. We assume the data domain D is equipped with some equality relation $=_D \subseteq D \times D$ (e.g., for C# we have $(x, y) \in =_D$ iff a

and b are of the same type and comparison by the `equals` method returns `true`). We denote by $\mathcal{E} = 2^{Val \times Val}$ the set of all relations over *Val*. We define a *default equality relation* $E_{def} \in \mathcal{E}$ as follows: We have $(x, y) \in E_{def}$ iff $x =?$ or $y =?$ or $(x, y) \in =_D$. We have $((\mathtt{f}, x_1, \ldots, x_n), (\mathtt{f}', y_1, \ldots, y_n)) \in E_{def}$ iff $\mathtt{f} = \mathtt{f}'$ and $(x_i, y_i) \in E_{def}$ for all $1 \leq i \leq n$.

Computation trace. A *(computation) trace* γ over some finite set of (programming) locations *Loc* is a finite sequence of location-value pairs $(Loc \times Val)^*$. We use the notation Γ_{Loc} to denote the set of all computation traces over *Loc*. Given some $\gamma \in \Gamma_{Loc}$ and $Loc' \subseteq Loc$, we denote by $\gamma|_{Loc'}$ the sequence that we obtain by deleting all pairs (ℓ, val) from γ, where $\ell \notin Loc'$.

3.2 Student Implementation

In the following we describe how a computation trace γ is generated for a student implementation Q on a given input σ. The computation trace is initialized to the empty sequence $\gamma = \epsilon$. Then the implementation is executed on σ according to the semantics of $\mathcal{L}$. During the execution we append location-value pairs to γ for every assignment statement: For $\ell : v_1 := e$ or $\ell : v_1[v_2] := e$ we append $(\ell, \sigma(v_1))$ to γ (we denote by $\sigma(v_1)$ the current value of v_1). We point out that we add the complete array $\sigma(v_1)$ to the trace for an assignment to an array variable v_1. For a library function call $\ell : v := \mathtt{f}(v_1, \ldots, v_n)$ we append $(\ell, (\mathtt{f}, \sigma(v), \sigma(v_1), \ldots, \sigma(v_n)))$ to γ. We denote the resulting trace γ by $[\![Q]\!](\sigma)$. This construction of a computation trace can be achieved by *instrumenting the implementation* in an appropriate manner.

3.3 Teacher Specification

The teacher uses `observe` and `observeFun` for specifying the key values she wants to observe during the execution of the specification and for defining an equality relation over computation domain. As usual the rectangular brackets '[' and ']' enclose optional arguments.

In the following we describe how a computation trace γ is generated for a specification P on a given input σ. The computation trace is initialized to the empty sequence $\gamma = \epsilon$. Then the specification is executed according to the semantics of $\mathcal{L}$. During the execution we append location-value pairs to γ only for `observe` and `observeFun` statements: For ℓ : `observe`$(v, [E])$ we append $(\ell, \sigma(v))$ to γ (we denote by $\sigma(v)$ the current value of v). For ℓ : `observeFun`$(\mathtt{f}[v_1, \ldots, v_n], [E])$ we append $(\ell, (\mathtt{f}, x_1, \ldots, x_n))$ to γ, where $x_i = \sigma(v_i)$, if the i^{th} argument to `f` has been specified, and $x_i =?$, if it has been left out. We denote the resulting trace γ by $[\![P]\!](\sigma)$.

Custom data equality. The possibility of specifying an equality relation $E \in \mathcal{E}$ at some location ℓ is very useful for the teacher. We point out that in practice the teacher has to specify E by an equality function $(Val \times Val) \to \{\mathtt{true}, \mathtt{false}\}$. The teacher can use E to define the equality of *similar computation values*. We show its usage on examples **R3** and **R4** (Fig. 1); both examples implement the *removing strategy* (discussed in §2) in almost identical ways — the only difference is on lines 10 and 9, respectively, where implementations use different characters to denote a character removed from a string: `'#'` and `' '`. In specification **RS** the teacher uses the equality function `CompareLetterString` (defined in **CDE**) — which compares only letters of two strings — to define value representations of both implementations, regardless of used characters, as equal.

We call a function $\delta : Loc \to \mathcal{E}$ a *comparison function*. We define $\delta(\ell) = E$ for every statement ℓ : `observe`(v, E) or ℓ : `observeFun`$(\mathtt{f}[v_1, \ldots, v_n], E)$. For statements, where $[E]$ has been left out, we set the default value $\delta(\ell) = E_{def}$.

Non-deterministic choice. We assume that the teacher can use some finite set of *non-deterministic* Boolean variables $B = \{nd_1, \ldots, nd_n\} \subseteq Var$ (these are not available to the student). Non-deterministic choice allows the teacher to specify variations in implementations, as discussed in §2. Non-deterministic variables are similar to the input variables, in the sense that are assigned before program is executed. We note that this results into 2^n different program behaviors for a given input.

4. MATCHING

In this section, we define what it means for an implementation to *(partially) match* or *fully match* a specification and describe the corresponding matching algorithms. The teacher has to determine for each specification which definition of matching has to be applied. In case of partial matching we speak of *inefficient specifications* and in case of full matching of *efficient specifications*.

4.1 Trace Embedding

We start out by discussing the problem of *Trace Embedding* that we use as a building block for the matching algorithms.

Subsequence. We call $c \in \{partial, full\}$ a *matching criterion*. Let $\gamma_1 = (\ell_1, val_1)(\ell_2, val_2) \cdots (\ell_n, val_n)$ and $\gamma_2 = (\ell'_1, val'_1)(\ell'_2, val'_2) \cdots (\ell'_m, val'_m)$ be two computation traces over some set of locations *Loc*, and let δ be some comparison function (as defined in §3.3). We say γ_1 is a *subsequence* of γ_2 w.r.t. to δ, c, written $\gamma_1 \sqsubseteq_{\delta,c} \gamma_2$, if there are indices $1 \leq k_1 < k_2 < \cdots < k_n \leq m$ such that for all $1 \leq i \leq n$ we have $\ell_i = \ell'_{k_i}$ and $(val_i, val'_{k_i}) \in \delta(\ell_i)$; in case of $c = full$ we additionally require that γ_1 and $\gamma_2|_{\{\ell_1, \ldots, \ell_n\}}$ have the same length. We refer to $(val_i, val'_{k_i}) \in \delta(\ell_i)$ as *equality check*. If $\delta(\ell_i) = Id$ (the identity relation over *Val*) for all $1 \leq i \leq n$, we obtain the usual definition of subsequence.

Since deciding subsequence, i.e., $\gamma_1 \sqsubseteq_{\delta,c} \gamma_2$, is a central operation in this paper, we state complexity of this decision problem. It is easy to see that deciding subsequence requires only $O(m)$ equality checks; basically one iteration over γ_2 is sufficient.

Mapping Function. Let Loc_1 and Loc_2 be two disjoint sets of locations. We call an injective function $\pi : Loc_1 \to Loc_2$ a *mapping function*. We lift π to a function $\pi : \Gamma_{Loc_1} \to \Gamma_{Loc_2}$ by applying it to every location, i.e., we set

$$\begin{aligned} \pi(\gamma) &= (\pi(\ell_1), val_1)(\pi(\ell_2), val_2) \cdots \\ \text{for } \gamma &= (\ell_1, val_1)(\ell_2, val_2) \cdots \in \Gamma_{Loc_1}. \end{aligned}$$

Given a comparison function δ, a matching criterion c, and computation traces $\gamma_1 \in \Gamma_{Loc_1}$ and $\gamma_2 \in \Gamma_{Loc_2}$ we say that γ_1 *can be embedded in* γ_2 by π, iff $\pi(\gamma_1) \sqsubseteq_{\delta \circ \pi^{-1}, c} \gamma_2$, and write $\gamma_1 \sqsubseteq^{\pi}_{\delta,c} \gamma_2$. We refer to π as *embedding witness*.

Executing a program on set of assignments I gives rise to a set of traces, one for each assignment $\sigma \in I$. We say that the set of traces $(\gamma_{1,\sigma})_{\sigma \in I}$ can be embedded in $(\gamma_{2,\sigma})_{\sigma \in I}$ by π iff $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\delta,c} \gamma_{2,\sigma}$ for all $\sigma \in I$.

Definition 1 (Trace Embedding). **Trace Embedding** *is the problem of deciding for given sets of traces* $(\gamma_{1,\sigma})_{\sigma \in I}$ *and* $(\gamma_{2,\sigma})_{\sigma \in I}$*, a comparison function* δ*, and a*

```
1: EMBED((γ_{1,σ})_{σ∈I}, (γ_{2,σ})_{σ∈I}, Loc_1, Loc_2, δ, c):
2:     G ← Loc_1 × Loc_2
3:     for all ℓ_1 ∈ Loc_1, ℓ_2 ∈ Loc_2:
4:         for all σ ∈ I:
5:             if γ_{1,σ}|_{ℓ_1} ⋤^{ℓ_1↦ℓ_2}_{δ,c} γ_{2,σ}|_{ℓ_2}:
6:                 G ← G \ {(ℓ_1, ℓ_2)}
7:                 break
8:     for all π ∈ MaximumBipartiteMatching(G):
9:         found ← true
10:        for all σ ∈ I:
11:            if γ_{1,σ} ⋤^π_{δ,c} γ_{2,σ}:
12:                found ← false
13:                break
14:        if found = true: return true
15:    return false
```

Figure 3: Algorithm for Trace Embedding problem.

matching criterion c, if there is a witness mapping function π, such that $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\delta,c} \gamma_{2,\sigma}$ for all $\sigma \in I$.

Complexity. Clearly, Trace Embedding is in NP (assuming equality checks can be done in polynomial time): we first guess the mapping function $\pi : Loc_1 \to Loc_2$ and then check $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\delta,c} \gamma_{2,\sigma}$ for all $\sigma \in I$ (which is cheap as discussed above). However, it turns out that Trace Embedding is NP-complete *even* for a *singleton* set I, a *singleton* computation domain *Val*, and the *full matching criterion*. We present a detailed proof, by reduction from Permutation Pattern [6], in our technical report [11].

Algorithm. Fig. 3 shows our algorithm, EMBED, for the Trace Embedding problem. A straightforward algorithmic solution for the trace embedding problem is to simply test all possible mapping functions. However, there is an *exponential number* of such mapping functions w.r.t. to the cardinality of Loc_1 and Loc_2. This exponential blowup seems unavoidable as the combinatorial search space is responsible for the NP hardness. The *core element* of our algorithm is a pre-analysis that narrows down the space of possible mapping functions effectively. We observe that if $\ell_2 = \pi(\ell_1)$ and $\gamma_1 \sqsubseteq^{\pi}_{\delta,c} \gamma_2$, then there exists a trace embedding restricted to locations ℓ_1 and ℓ_2, formally: $\gamma_1|_{\{\ell_1\}} \sqsubseteq^{\{\ell_1 \mapsto \ell_2\}}_{\delta,c} \gamma_2|_{\{\ell_2\}}$. The algorithm uses this insight to create a (bipartite) graph $G \subseteq Loc_1 \times Loc_2$ of potential mapping pairs in lines 2-7. A pair of locations $(\ell_1, \ell_2) \in G$ is a *potential mapping pair* iff there exists a trace embedding restricted to locations ℓ_1 and ℓ_2, as described above.

The key idea in finding an embedding witness π is to construct a *maximum bipartite matching* in G. A maximum bipartite matching has an edge connecting every program location from Loc_1 to a distinct location in Loc_2 and thus gives rise to an injective function π. We point out that such an injective function π does not need to be an embedding witness, because, by observing only a single location pair at a time, it ignores the order of locations. Thus, for each maximum bipartite matching [27] π the algorithm checks (in lines 8-14) if it is indeed an embedding witness.

The key strength of our algorithm is that it reduces the search space for possible embedding witnesses π. The experimental evidence shows that this approach significantly reduces the number of possible matchings and enables a very efficient algorithm in practice, as discussed in §6.

```
1: MATCHES(Specification P, Implementation Q, Inputs I):
2:     Loc_1 = observed locations in P
3:     δ = comparison function specified by P
4:     c = matching criterion
5:     Loc_2 = assignment locations of Q
6:     for all σ ∈ I:
7:         γ_{2,σ} ← [[Q]](σ)
8:     B_P = non-deterministic variables in P
9:     for all assignments σ_nd to B_P:
10:        for all σ ∈ I:
11:            γ_{1,σ} ← [[P]](σ ∪ σ_nd)
12:        if EMBED((γ_{1,σ})_{σ∈I}, (γ_{2,σ})_{σ∈I}, Loc_1, Loc_2, δ, c):
13:            return true
14:    return false
```

Figure 4: Matching algorithm.

4.2 Partial Matching

We now define the notion of *partial matching* (also referred to simply as *matching*) which is used to check whether an implementation involves (at least) those inefficiency issues that underlie a given inefficient specification.

DEFINITION 2 (PARTIAL MATCHING). *Let P be a specification with observed locations Loc_1, let δ be the comparison function specified by P, and let Q be an implementation whose assignment statements are labeled by Loc_2. Then implementation Q (partially)* **matches** *specification P, on a set of inputs I, if and only if there exists a mapping function $\pi : Loc_1 \to Loc_2$ and an assignment to the non-deterministic variables σ_{nd} such that $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\delta,c} \gamma_{2,\sigma}$, for all input values $\sigma \in I$, where $\gamma_{1,\sigma} = [\![P]\!](\sigma \cup \sigma_{nd})$, $\gamma_{2,\sigma} = [\![Q]\!](\sigma)$ and $c = partial$.*

Fig. 4 describes an algorithm for testing if an implementation (partially) matches a given specification over a given set of input valuations I. In lines 6-7, the implementation Q is executed on all input values $\sigma \in I$. In line 9, the algorithm iterates through all assignments σ_{nd} to the non-deterministic variables B_P of the specification P. In lines 10-11, the specification P is executed on all inputs $\sigma \in I$. With both sets of traces available, line 12 calls subroutine EMBED which returns `true` iff there exists a trace embedding witness.

Example. We now give an example that demonstrates our notion of programs and that contains example applications of algorithms EMBED and MATCHES. In Fig. 5 we state two implementations, (a) and (b), and one specification (c). These programs represent simplified versions (transformed into three adress code) of **R1** (after function inlining), **R3** and **SC** (Fig. 1). Note, that every assignment and `observe` statement is on its own line; we denote line i in program x by by location $\ell_{x,i}$. The argument $[E]$ has been left out for all locations in the specification, thus we have $\delta(\ell) = E_{def}$ for all specification locations ℓ.

Algorithm MATCHES runs all three programs on input values s = "aab" and t = "aba". For program (a) we obtain the following computation trace:
$\gamma_a = (\ell_{a,2}, 0)(\ell_{a,3}, 3)(\ell_{a,5}, a)(\ell_{a,6}, 0)(\ell_{a,7}, 0)(\ell_{a,9}, a)(\ell_{a,11}, 1)$
$(\ell_{a,12}, 1)(\ell_{a,9}, a)(\ell_{a,11}, 2)(\ell_{a,12}, 2)(\ell_{a,9}, b)(\ell_{a,12}, 3)(\ell_{a,13}, 0) \cdots$
Similarly, for program (b) we obtain:
$\gamma_b = (\ell_{b,2}, 0)(\ell_{b,3}, 3)(\ell_{b,5}, a)(\ell_{b,6}, (\text{SPLIT}, aab, a))(\ell_{b,7}, 3)$
$(\ell_{b,8}, (\text{SPLIT}, aba, a))(\ell_{b,9}, 3)(\ell_{b,10}, 1)(\ell_{b,5}, a) \cdots$
For specification (c) we obtain two traces, depending on the

(a)

```
1 Puzzle(s, t) {
2   i = 0;
3   n = |s|;
4   while (i < n) {
5     c = s[i];
6     j = 0;
7     cnt1 = 0;
8     while (j < n) {
9       c2 = s[j];
10      if (c == c2) {
11        cnt1 = cnt1 + 1; }
12      j = j + 1; }
13    j = 0;
14    cnt2 = 0;
15    while (j < n) {
16      c2 = t[j];
17      if (c == c2) {
18        cnt2 = cnt2 + 1; }
19      j = j + 1; }
20    i = i + 1; }}
```

(b)

```
1 Puzzle(s, t) {
2   i = 0;
3   n = |s|;
4   while (i < n) {
5     c = s[i];
6     ss = Split(s,c);
7     cnt1 = |ss|;
8     st = Split(t,c);
9     cnt2 = |st|;
10    i = i + 1; }}
```

(c)

```
1  Puzzle(s, t) {
2    i = 0;
3    n = |s|;
4    while (i < n) {
5      c = s[i];
6      observe(c);
7      j = 0;
8      cnt1 = 0;
9      while (j < n) {
10       c2 = s[j];
11       if (nd1)
12         observe(c2);
13       j = j + 1; }
14     j = 0;
15     if (!nd1)
16       observeFun(Split());
17     while (j < n) {
18       c2 = t[j];
19       if (nd1)
20         observe(c2);
21       j = j + 1; }
22     if (!nd1)
23       observeFun(Split());
24     i = i + 1; }}
```

Figure 5: Implementations (a), (b) and Spec. (c).

choice for the non-deterministic variable *nd1*:
$\gamma_{c,\mathtt{t}} = (\ell_{c,6}, a)(\ell_{c,12}, a)(\ell_{c,12}, a)(\ell_{c,12}, b)(\ell_{c,20}, a)(\ell_{c,20}, b)\cdots$
$\gamma_{c,\mathtt{f}} = (\ell_{c,6}, a)(\ell_{c,16}, (\textsc{Split}, ?, ?))(\ell_{c,23}, (\textsc{Split}, ?, ?))\cdots$

Algorithm Matches then calls Embed to check for trace embedding. Algorithm Embed first constructs a potential graph G, which contains an edge for two locations of the specification and the implementation that show the same values.

For implementation (a), we obtain the following graph: $G_a = \{(\ell_{c,6}, \ell_{a,5}), (\ell_{c,6}, \ell_{a,9}), (\ell_{c,6}, \ell_{a,16}), (\ell_{c,12}, \ell_{a,9}), (\ell_{c,20}, \ell_{a,16})\}$. Notice that $\ell_{c,6}$ shows the same values as the locations $\ell_{a,5}, \ell_{a,9}, \ell_{a,16}$ in the implementation (a). However, there is only one maximal matching in G_a, $\pi_a = \{(\ell_{c,6}, \ell_{a,5}), (\ell_{c,12}, \ell_{a,9}), (\ell_{c,20}, \ell_{a,16})\}$, which is also an embedding witness; thus implementation (a) matches specification (c).

For implementation (b) and *nd1* = `true`, we obtain the graph $G_{b,\mathtt{t}} = \{(\ell_{c,6}, \ell_{b,5})\}$, from which we cannot construct a maximal matching. However, for *nd1* = `false`, we obtain $G_{b,\mathtt{f}} = \{(\ell_{c,6}, \ell_{b,5}), (\ell_{c,16}, \ell_{b,6}), (\ell_{c,23}, \ell_{b,8})\}$, which is also an embedding witness; thus implementation (b) matches specification (c).

4.3 Full Matching

Below we will define the notion of *full matching*, which is used to match implementations against efficient specifications. We will require that for every loop and every library function call in the implementation there is a corresponding loop and library function call in the matching specification. In order to do so, we need some helper definitions.

Observed loop iterations. We extend the construction of the implementation trace (defined in §3.2): For each statement ℓ : `while` v `do` s, we additionally append element $(\ell, \bot)$ to the trace whenever the loop body s is entered. We call $(\ell, \bot)$ a *loop iteration*. Let π be a embedding witness s.t., $\gamma_1 \sqsubseteq^{\pi} \gamma_2$. We say that π *observes all loop iterations* iff between every two loop iterations $(\ell, \bot)$ in γ_2 there exists a pair (ℓ', val), such that $\exists \ell''.\pi(\ell'') = \ell'$. In other words, we require that between any two iterations of the same loop, there exists some observed location ℓ'.

Observed library function calls. We say that π *observes all library function calls* iff for every $(\ell, \mathtt{f}(val_1, \ldots, val_n))$ in γ_2 there is a ℓ' such that $\pi(\ell') = \ell$.

Definition 3 (Full Matching). *Let P be a specification with observed locations Loc_1, let δ be the comparison function specified by P, and let Q be an implementation whose assignment statements are labeled by Loc_2. Then implementation Q* **fully matches** *specification P, on a set of inputs I, if and only if there exists a mapping function $\pi : Loc_1 \to Loc_2$ and an assignment to the non-deterministic variables σ_{nd} such that $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\delta,c} \gamma_{2,\sigma}$, for all input valuations $\sigma \in I$, where $\gamma_{1,\sigma} = [\![P]\!](\sigma \cup \sigma_{nd})$, $\gamma_{2,\sigma} = [\![Q]\!](\sigma)$, $c = full$ and π observes all loop iterations and library function calls.*

We note that procedure Embed (Fig. 3) can easily check at line 11 whether the current mapping π observes all loop iterations and library function calls.

It is tedious for a teacher to *exactly specify* all possible loop iterations and library function calls used in different efficient implementations. We add two additional constructs to the language $\mathcal{L}$ to simplify this specification task.

Cover statement. We extend $\mathcal{L}$ by two *cover statements*: ℓ : `cover`$(\mathtt{f}[v_1, \ldots, v_m], [E])$ and ℓ : `cover`(v). The first statement is the same as the statement ℓ : `observeFun`$(\mathtt{f}[v_1, \ldots, v_m], [E])$, except that we allow the embedding witness π to not map ℓ to any location in the implementation. This enables the teacher to specify that function $\mathtt{f}(v_1, \ldots, v_m)$ *may appear* in the implementation. The second statement allows π to map ℓ to a location ℓ' that appears *at most* $\sigma(v)$ times for each appearance of ℓ, where $\sigma(v)$ is the current value of the specified variable v. Thus `cover`(v) enables the teacher to *cover any loop* with up to $\sigma(v)$ iterations.

Example. Now we present examples for efficient implementations (**E1** and **E2**) and specification (**ES**) for the Anagram problem (Fig. 1). The teacher observes computed values on lines 9, 12 and 14, and uses a non-deterministic choice (on line 11) to choose if implementations count the number of characters in each string, or decrement one number from another. Also the teacher allows *up to* two library function calls and two loops with *at most* 255 iterations, defined by `cover` statements on lines 4,5,6 and 17.

5. EXTENSIONS

In this section, we discuss useful extensions to the core material presented above. These extensions are part of our implementation, but we discuss them separately to make the presentation easier to follow.

One-to-many Mapping. According to definition of Trace Embedding, an embedding witness π maps one implementation location to a specification location, i.e., it constructs a *one-to-one* mapping. However, it is possible that a student *splits* a computation of some value over multiple locations. For example, in the implementation stated in **R5** (Fig. 1), the student removes a character from a string across three different locations (on lines 9, 11, 13 and 14), depending on the location of the removed character in the string. This requires to map a *single* location from the specification to *multiple* locations in the implementation! For this reason,

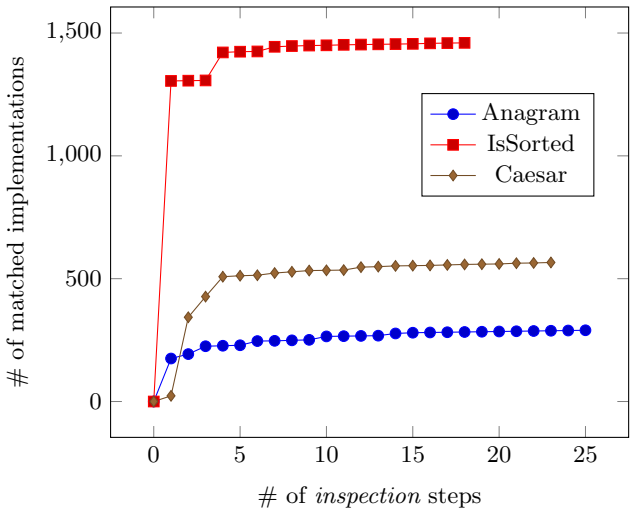

Figure 6: The number of inspection steps required to completely specify the MOOC-style assignments.

we extend the notion of trace embedding to *one-to-many* mappings $\pi : Loc_1 \to 2^{Loc_2}$ where $\pi(\ell') \cap \pi(\ell) = \emptyset$ for all $\ell \neq \ell'$. It is easy to extend procedure EMBED (Fig. 3) to this setting: the potential graph G is also helpful to enumerate every possible *one-to-many* mapping. However, it is costly (and unnecessary) to search for arbitrary one-to-many mappings. We use heuristics to consider only a few one-to-many mappings. For example, one of the heuristics in our implementation checks if the same variable is assigned in different branches of an if-statement (e.g., in example **R5**, for all three locations there is an assignment to variable *cp*).

Although *many-to-many* mappings may seem more powerful, we point out that the teacher can always write a specification that is more succinct than the implementation of the student, i.e., the above described one-to-many mappings provide enough expressivity to the teacher.

Non-deterministic behaviour. In our technical report [11] we discuss another extension: libraries with non-deterministic behaviour (e.g., the iteration order over a set).

6. IMPLEMENTATION AND EXPERIMENTS

We now describe our implementation and present an experimental evaluation of our framework. More details on our experiments can we found on the website [1].

6.1 Experimental Setup

Our implementation of algorithm MATCHES (Fig. 4) is in C# and analyzes C# programs (i.e., implementations and specifications are in C#). We used Microsoft's Roslyn compiler framework [3] for instrumenting every sub-expression to record value during program execution.

Data. We used 3 preexisting problems from PEX4FUN (as mentioned in §1): (1) the *Anagram* problem, where students are asked to test if two strings could be permuted to become equal, (2) the *IsSorted* problem, where students are asked to test if the input array is sorted, and (3) the *Caesar* problem, where students are asked to apply Caesar cipher to the input string. We have chosen these 3 specific problems because they had a high number of student attempts, diversity in algorithmic strategies and a problem was explicitly stated (for many problems on PEX4FUN platform students have to guess the problem from failing input-output examples).

We also created a new course on the PEX4FUN platform with 21 programming problems. These problems were assigned as a homework to students in a second year undergraduate course. We created this course to understand performance related problems that *CS* students make, as opposed to regular PEX4FUN users who might not have previous programming experience. We encouraged our students to write efficient implementations by giving more points for performance efficiency than for mere functional correctness. We omit the description of the problems here, but all descriptions are available on the original course page [2].

6.2 Methodology

In the following we describe the methodology by which we envision the technique in the paper to be used.

The teacher maintains a set of efficient and inefficient specifications. A new student implementation is checked against all available specifications. If the implementation matches some specification, the associated feedback is automatically provided to the student; otherwise the teacher is notified that there is a new unmatched implementation. The teacher studies the implementation and identifies one of the following reasons for its failure to match any existing specification: (i) The implementation uses a new strategy not seen before. In this case, the teacher creates a new specification. (ii) The existing specification for the strategy used in the implementation is too specific to capture the implementation. In this case, the teacher refines that existing specification. This overall process is repeated for each unmatched implementation.

New specification. A teacher creates a new specification using the following steps: (i) Copy the code of the unmatched implementation. (ii) Annotate certain values and function calls with observe statements. (iii) Remove any unnecessary code (not needed in the specification) from the implementation. (iv) Identify input values for the dynamic analysis for matching. (v) Associate a feedback with the specification.

Specification refinement. To refine a specification, the teacher identifies one of the following reasons as to why an implementation did not match it: (i) The implementation differs in details specified in the specification; (ii) The specification observes more values than those that appear in the implementation; (iii) The implementation uses different data representation. In case (i) the teacher adds a new non-deterministic choice, and, if necessary, observes new values or function calls; in case (ii) the teacher observes less values; and in case (iii) the teacher creates or refines a custom data-equality.

Input values. Our dynamic analysis approach requires the teacher to associate input values with specifications. These input values should cause the corresponding implementations to exhibit their worst-case behavior; otherwise an inefficient implementation might behave similar to an efficient implementation and for this reason match the specification of the efficient implementation. This implies that *trivial* inputs should be avoided. For example, two strings with unequal lengths constitute a trivial input for the counting strategy since each of its three implementations **C1**-**C3** (Fig. 1) then exit immediately. Similarly, providing a sorted input for the sorting strategy is meaningless. We remark that it is easy for a teacher (who understands the various strategies) to provide good input values.

Problem Name	Correct Implement.	Inefficient Implement.	N	S	I	ND	$^{L_S}/_{\overline{L_I}}$	O_S	O_I	M	Performance Avg.	Performance Max.
Anagram	290 (37.9%)	261 (90.0%)	5	25	1	3	1.41	11	89	28357	0.42	7.67
IsSorted	1460 (90.1%)	139 (9.5%)	3	23	2	2	1.45	6	51	13	0.33	1.31
Caesar	566 (81.2%)	343 (60.6%)	5	18	1	1	1.10	7	39	172	0.37	0.83
DoubleChar	46 (97.9%)	31 (67.4%)	1	5	1	0	0.72	3	23	2	0.31	0.42
LongestEqual	37 (78.7%)	1 (2.7%)	1	3	1	0	0.57	1	35	2	0.33	0.44
LongestWord	39 (83.0%)	13 (33.3%)	2	6	2	0	1.31	7	46	15	0.35	0.47
RunLength	43 (97.7%)	32 (74.4%)	1	6	1	0	0.90	8	37	54	0.33	0.44
Vigenere	41 (93.2%)	32 (78.0%)	3	5	1	0	0.64	3	84	6	0.34	0.50
BaseToBase	15 (39.5%)	14 (93.3%)	2	5	1	1	0.35	3	64	13	0.36	0.48
CatDog	41 (87.2%)	8 (19.5%)	2	18	1	1	2.02	21	53	1629	0.36	0.58
MinimalDelete	15 (39.5%)	8 (53.3%)	1	8	2	3	2.21	4	75	10	0.86	4.36
CommonElement	43 (95.6%)	32 (74.4%)	4	14	2	1	0.97	6	79	107	0.36	0.53
Order3	40 (87.0%)	30 (75.0%)	6	12	1	2	1.45	6	78	19	0.40	0.59
2DSearch	37 (84.1%)	36 (97.3%)	3	7	1	1	1.09	2	67	1	0.34	0.45
TableAggSum	11 (25.0%)	10 (90.9%)	1	5	1	1	0.80	3	144	1	0.40	0.53
Intersection	14 (31.8%)	12 (85.7%)	3	7	2	1	0.89	4	73	5	0.37	0.56
ReverseList	39 (97.5%)	0 (0.0%)	0	3	1	0	0.35	4	34	1	0.34	0.44
SortingStrings	41 (91.1%)	34 (82.9%)	5	11	1	1	1.48	13	110	866	0.55	14.59
MinutesBetween	45 (100.0%)	0 (0.0%)	0	5	1	0	0.64	8	101	1	0.37	0.48
MaxSum	42 (95.5%)	17 (40.5%)	2	7	1	1	1.14	2	51	3	0.35	0.47
Median	47 (100.0%)	47 (100.0%)	1	1	1	0	0.39	1	100	1	0.34	0.44
DigitPermutation	36 (100.0%)	1 (2.8%)	1	3	1	0	0.26	4	29	4	0.32	0.44
Coins	27 (65.9%)	14 (51.9%)	2	6	1	1	1.65	4	93	175	2.41	15.44
Seq235	33 (89.2%)	30 (90.9%)	4	12	1	2	1.79	3	232	3	0.94	22.08

Table 1: List of all assignments with the experimental results.

Granularity of feedback. We want to point out that the granularity of a feedback depends on the teacher. For example, in a programming problem where sorting the input value is an inefficient strategy, the teacher might not want to distinguish between different sorting algorithms, as they do not require a different feedback. However, in a programming problem where students are asked to implement a sorting algorithm it makes sense to provide a different feedback for different sorting algorithms.

6.3 Evaluation

We report results on the 24 problems discussed above in Table 1.

Results from manual code study. We first observe that a large number of students managed to write a functionally correct implementation on most of the problems (column *Correct Implementations*). This shows that Pex4Fun succeeds in guiding students towards a correct solution.

Our second observation is that for most problems a large fraction of implementations is inefficient (column *Inefficient Implementations*), especially for *Anagram* problem: 90%. This shows that although students manage to achieve functional correctness, efficiency is still an issue (recall that in our homework the students were explicitly asked and given extra points for efficiency).

We also observe that for all, except two, problems there is at least one inefficient algorithmic strategy, and for most problems (62.5%) there are several inefficient algorithmic strategies (column N). *These results highly motivate the need for a tool that can find inefficient implementations and also provide a meaningful feedback on how to fix the problem.*

Precision and Expressiveness. For each programming assignment we used the above described methodology and wrote a specification for each algorithmic strategy (both efficient and inefficient). We then *manually verified* that each specification matches all implementations of the strategy, hence providing desired feedback for implementations. *This shows that our approach is* precise *and* expressive *enough to capture the algorithmic strategy, while ignoring low level implementation details.*

Teacher Effort. To provide manual feedback to students the teacher would have to go through every implementation and look at its performance characteristics. In our approach the teacher has to take a look only at a few representative implementations. In column S we report the total number of inspection steps that we required to fully specify one programming problem, i.e., the number of implementations that the teacher would had to go through to provide feedback on all implementations. For the 3 pre-existing problems *the teacher would only have to go through 66 out of 2316 (or around 3%) implementations to provide full feedback.* Fig. 6 shows the number of matched implementations with each inspection step (by inspecting the first unmatched implementation in the random order). For space reasons the diagram shows only 3 pre-existing assignments; our technical report [11] shows all 24 assignments as well as the time it took us for each inspection step.

In column $^{L_S}/_{\overline{L_I}}$we report the largest ratio of specification and average matched implementation in terms of lines of code. We observe that in half of the cases the largest specification is about the same size or smaller than the average matched implementation. Furthermore, the number of the input values that need to be provided by the teacher is 1-2 across all problems (column I). In all but one problem (*IsSorted*) one set of input values is used for all specifications. Also, in about one third of the specifications there was no need for non-deterministic variables, and the largest number used in one specification is 3 (column *ND*). *Overall, our semi-automatic approach requires considerably less teacher effort than providing manual feedback.*

Performance. We plan to integrate our framework in a MOOC platform, so performance, as for most web applications, is critical. Our implementation consists of two parts. The first part is the execution of the implementation and

the specification (usually small programs) on relatively small inputs and obtaining execution traces, which is, in most cases, neglectable in terms of performance. The second part is the EMBED algorithm. As discussed in §4.1 the challenge consists in finding an embedding witness π. With O_S observed variables in the specification and O_I observed variables in the implementation, there are $\frac{O_I!}{(O_I-O_S)!}$ possible injective mapping functions. E.g., for the *SortingStrings* problem that gives $\approx 10^{26}$ possible mapping functions ($O_I = 110, O_S = 13$). However, our algorithm reduces this huge search space by constructing a bipartite graph G of potential mappings pairs. In M we report the number of mapping functions that our tool had to explore. E.g., for *SortingStrings* only 866 different mapping functions had to be explored. For all values (O_S, O_I and M) we report the maximal number across all specifications. In the last column we state the *total execution time* required to decide if one implementation matches the specification (average and maximal). Note that this time includes execution of both programs, exploration of all assignments to non-deterministic Boolean variables and finding an embedding witness π. Our tool runs, in most cases, under half a second per implementation. *These results show that our tool is fast enough to be used in an interactive teaching environment.*

6.4 Threats to Validity

Unsoundness. Our method is unsound in general since it uses a dynamic analysis that explores only a few possible inputs. However, we did not observe any unsoundness in our large scale experiments. If one desires provable soundness, an embedding witness could be used as a guess for a simulation relation that can then be formally verified by other techniques. Otherwise, a student who suspects an incorrect feedback can always bring it to the attention of the teacher.

Program size. We evaluated our approach on introductory programming assignments. Although questions about applicability to larger programs might be raised, our goal was not to analyze arbitrary programs, but rather to develop a framework to help teachers who teach introductory programming with providing performance feedback — currently a manual, error-prone and time-consuming task.

Difficulty of the specification language. Although we did not perform case study with third-party instructors, we report our experiences with using the proposed language. We would also like to point out that writing specifications is a one-time investment, which could be performed by an experienced personnel.

7. RELATED WORK

7.1 Automated Feedback

There has been a lot of work in the area of generating automated feedback for programming assignments. This work can be classified along three dimensions: (a) aspects on which the feedback is provided such as functional correctness, performance characteristics or modularity (b) nature of the feedback such as counterexamples, bug localization or repair suggestions, and (c) whether static or dynamic analysis is used.

Ihantola et.al. [14] present a survey of various systems developed for automated grading of programming assignments. The majority of these efforts have focussed on checking for functional correctness. This is often done by examining the behavior of a program on a set of test inputs. These test inputs can be manually written or automatically generated [26]. There has only been little work in testing for non-functional properties. The ASSYST system uses a simple form of tracing for counting execution steps to gather performance measurements [15]. The Scheme-robo system counts the number of evaluations done, which can be used for very coarse complexity analysis. The authors conclude that better error messages are the most important area of improvement [21].

The AI community has built tutors that aim at bug localization by comparing source code of the student and the teacher's programs. LAURA [5] converts teacher's and student's program into a graph based representation and compares them heuristically by applying program transformations while reporting mismatches as potential bugs. TALUS [18] matches a student's attempt with a collection of teacher's algorithms. It first tries to recognize the algorithm used and then tentatively replaces the top-level expressions in the student's attempt with the recognized algorithm for generating correction feedback. In contrast, we perform trace comparison (instead of source code comparison), which provides robustness to syntactic variations.

Striewe and Goedicke have proposed localizing bugs by trace comparisons. They suggested creating full traces of program behavior while running test cases to make the program behavior visible to students [23]. They have also suggested automatically comparing the student's trace to that of a sample solution [24] for generating more directed feedback. However, no implementation has been reported. We also compare the student's trace with the teacher's trace, but we look for similarities as opposed to differences.

Recently it was shown that automated techniques can also provide repair based feedback for functional correctness. Singh's SAT solving based technology [22] can successfully generate feedback (of up to 4 corrections) on around 64% of all incorrect solutions (from an MIT introductory programming course) in about 10 seconds on average. While test inputs provide guidance on *why* a given solution is incorrect and bug localization techniques provide guidance on *where* the error might be, repairs provide guidance on *how* to fix an incorrect solution. We also provide repair suggestions that are manually associated with the various teacher specifications, but for performance based aspects. Furthermore, our suggestions are not necessarily restricted to small fixes.

7.2 Performance Analysis

The Programming Languages and Software Engineering communities have explored various kinds of techniques to generate performance related feedback for programs. Symbolic execution based techniques have been used for identifying non-termination related issues [13, 7]. The SPEED project investigated use of static analysis techniques for estimating symbolic computational complexity of programs [28, 10, 12]. Goldsmith et.al. used dynamic analysis techniques for empirical computational complexity [8]. The Toddler tool reports a specific pattern: computations with repetitive and similar memory-access patterns [20]. The Cachetor tool reports memoization opportunities by identifying operations that generate identical values [19]. In contrast, we are interested in not only identifying whether or not there is a performance issue, but also identifying its root cause and generating repair suggestions.

8. REFERENCES

[1] http://forsyte.at/static/people/radicek/fse14.

[2] Making Programs Efficient. http://pexforfun.com/makingprogramsefficient.

[3] Microsoft "Roslyn" CTP. http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx.

[4] Pex for fun. http://www.pexforfun.com/.

[5] A. Adam and J.-P. H. Laurent. LAURA, a system to debug student programs. *Artif. Intell.*, 15(1-2), 1980.

[6] P. Bose, J. F. Buss, and A. Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998.

[7] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, pages 161–169, 2009.

[8] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, 2007.

[9] S. Gulwani. Example-based learning in computer-aided STEM education. *To appear in Commun. ACM*, 2014.

[10] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[11] S. Gulwani, I. Radiček, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. *CoRR*, abs/1403.4064, 2014.

[12] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[13] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

[14] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

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

[16] K. Masters. A brief guide to understanding MOOCs. *The Internet Journal of Medical Education*, 1(2), 2011.

[17] R. Milner. An algebraic definition of simulation between programs. Technical report, Stanford, CA, USA, 1971.

[18] W. R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3, 1987.

[19] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM.

[20] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[21] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 133–136, New York, NY, USA, 2001. ACM.

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

[23] M. Striewe and M. Goedicke. Using run time traces in automated programming tutoring. In *ITiCSE*, pages 303–307, 2011.

[24] M. Striewe and M. Goedicke. Trace alignment for automated tutoring. In *CAA*, 2013.

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

[26] N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *ICSE*, 2013.

[27] T. Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *ISAAC*, pages 92–101, 1997.

[28] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.