Automated Semantic Grading of Programs

Rishabh Singh
MIT CSAIL, Cambridge, MA
rishabh@csail.mit.edu

Sumit Gulwani
Microsoft Research, Redmond, WA
sumitg@microsoft.com

Armando Solar-Lezama
MIT CSAIL, Cambridge, MA
asolar@csail.mit.edu

Abstract

We present a new method for automatically grading introductory programming assignments. In order to use this method, instructors provide a reference implementation of the assignment, and an error model consisting of potential corrections to errors that students might make. Using this information, the system automatically derives minimal corrections to student's incorrect solutions, providing them with a quantifiable measure of exactly how incorrect a given solution was, as well as feedback about what they did wrong.

We introduce a simple language for describing error models in terms of correction rules, and formally define a rule-directed translation strategy that reduces the problem of finding minimal corrections in an incorrect program to the problem of synthesizing a correct program from a sketch. We have evaluated our system on over 1000 solution attempts by real beginner programmers. Our results show that relatively simple error models can correct on average 73% of fixable fraction of submissions with non-trivial errors. We also show that the error models generalize across different problems from the same category, and our technique scales well for more complex error models and programming assignments such as those found in AP level computer science final examinations.

1. Introduction

There has been a lot of recent interest in making education more accessible through information technology. Recently, for example, Sebastian Thurn and Peter Norvig at Stanford made headlines when they allowed everyone in the world to register for their class and take it over the internet. One of the big bottlenecks in making this approach work, however, is the question of providing feedback to students, particularly when it comes to exams and assignments; in the case of the Stanford course, for example, 50,000 students turned in the first assignment[1]. Providing personalized human feedback for that many assignments is prohibitively expensive, so Norvig and Thurn opted for not grading programming assignments; instead, students are assigned a grade by entering a few outputs from their program into a web form. The problem with this approach is that if their solution is not correct, the student does not get any feedback about what went wrong or how to fix it. The most a student gets in terms of feedback is a pointer to a video specifically recorded to help students who got the assignment wrong[1].

Automated test-cases based grading is one of the most commonly used method today for grading programming assignments but unfortunately it is not ideal, especially for beginner programmers, as the only feedback they get is a set of failing test cases. We have observed many instances in our data where such feedback of failing test cases did not help students understand their mistakes and eventually led them to give up. In fact, for the Introduction to Programming course taught at MIT (`6.00`), the Teaching Assistants are required to manually go over each submitted assignment and provide personalized feedback. This method of manual grading is a time consuming process – grading an assignment for a class of 200 students takes approximately a week. With online education initiatives like MITx and Udacity expecting student registrations of over 100,000 for their introductory programming courses, performing manual grading becomes impractical. In this paper, we present a complimentary approach for automatically grading a student's solution that also provides feedback describing exactly what was wrong with the solution.

The approach leverages program synthesis technology to automatically determine minimal fixes to the student's solution that will make it match the behavior of a reference solution written by the instructor. This technology provides a basis for assigning partial credit for incorrect solutions by giving a quantitative measure of the degree to which the solution was incorrect, and makes it possible to provide students with precise feedback about what they did wrong.

The problem of providing automatic feedback is closely related to the problem of automated bug fixing, but it differs from it in following three important respects:

- **The complete specification is known.** An important challenge in automatic debugging is that there is no way to know whether a fix is addressing the root cause of a problem, or simply masking it and potentially introducing new errors. Usually the best one can do is check a candidate fix against a test suite or a partial specification [9]. While grading on the other hand, the solution to the problem is known, and it is safe to assume that the instructor already wrote a correct reference implementation for the problem.
- **Errors are predictable.** In a homework assignment, everyone is solving the same problem after having attended the same lectures, so errors tend to follow predictable patterns. This makes it possible to use a *model-based* grading approach, where the potential fixes are guided by a model of the kinds of errors students typically make.
- **Programs are small.** This is particularly true for introductory programming assignments, where students are learning basic programming constructs as well as simple algorithms like sorting. This makes the problem fundamentally different from the problem of debugging large-scale software by allowing use of expensive but powerful algorithms.

These simplifying assumptions, however, introduce their own set of challenges. For example, while the reference implementation is known, the student solution might be completely different from it, so the grading tool needs to be able to reason about the equivalence of the student solution with a reference implementation. Also, in order to take advantage of the predictability of errors, the tool needs to provide a simple mechanism to describe the classes of errors. And finally, while the programs are small, they can be expected to

[1] http://www.stanford.edu/group/knowledgebase/cgi-bin/2011/10/21/stanfords-online-engineering-classes-hugely-popular/

```
1 using System;

3 public class Program {
4   public static int[] Puzzle(int[] b) {
5     int front, back, temp;
6     front = 0;
7     back = b.Length-1;
8     temp = b[back];

10    while (front > back){
11      b[back] = b[front];
12      b[front] = temp;
13      back++;
14      front++;
15      temp = b[back];
16    }
17    return b;
18  }
19 }
```

(a)

```
1 using System;

3 public class Program {
4   public static int[] Puzzle(int[] b) {
5     for (int i=0; i <= b.Length/2; i++){
6       int temp = b[i];
7       b[i] = b[b.Length-i-1];
8       b[b.Length-i-1] = temp;
9     }
10    return b;
11  }
12 }
```

(b)

Figure 1. (a) A student's solution and (b) the instructor's reference implementation for the array reverse problem.

have higher density of errors than production code, so techniques like the one suggested by [23], which attempts to correct bugs one path at a time will not work for many of these problems that require coordinated fixes in multiple places.

Our automated grading technique handles all of these challenges. The tool can reason about the semantic equivalence of programs and reference implementations written in a subset of C#, so the instructor does not have to learn a new formalism to write specifications. The tool also provides an *error model* language that instructors can use to write an error model: a very high level description of potential corrections to errors that students might make in the solution. When the system encounters an incorrect solution by a student, it symbolically explores the space of all possible combinations of corrections allowed by the error model and finds a correct solution requiring a minimal set of corrections.

In order to evaluate our approach, we leveraged the PEX4FUN website[2]. PEX4FUN is a popular website maintained by the Pex team at Microsoft Research as a way to showcase their automated testing technology. The website allows instructors to submit programming problems and puzzles, which are routinely solved by users from all over the world. When a user submits a candidate solution, the site applies automated testing technology to find test inputs for which the submission fails. The website records all solution attempts by users, and distinguishes between attempts from different users, so it is possible to use the website logs to observe how users struggle to solve the programming problems and to see exactly what errors they make even when given feedback in the form of test input-output pairs. While we have no way of knowing exactly who the users of the website are (the logs are anonymized), it is safe to assume many of them are beginner programmers given the kind of mistakes they make on the programming problems, so we often refer to them as students or beginners through the rest of the paper.

As part of our evaluation, we analyzed over a thousand solution attempts to problems including loops-over-arrays problems such as array reverse, recursion problems such as factorial, as well as more complex problems from Advanced Placement (AP) computer science exams[3]. We observe that PEX4FUN users tend to repeat similar mistakes when solving these problems and that our error models on average can correct over 73% of submissions with small errors (`Effectiveness`). We also show that the error models generalize well across different problems from the same category and that our technique scales well even for complex error models and bigger programming assignments such as those found in AP examinations.

This paper makes the following key contributions:

- We show that the problem of automated grading and providing feedback for introductory programming assignments can be framed as a synthesis problem and be successfully solved using constraint-based synthesis technology.
- We define a high-level language EML that instructors can use to provide correction rules to be used for automated grading. We also show that a small set of such rules is sufficient to correct hundreds of incorrect solutions written by real beginner programmers.
- We present a technique to compute minimum cost corrections to a student solution based on the correction rules provided by an instructor.
- We report the successful evaluation of our technique on over 1000 solution attempts by real beginner programmers on problems ranging from introductory programming problems to problems from AP level exams.

2. Overview of the approach

In order to illustrate the key ideas behind our approach, consider the problem of reversing an array. The problem is simple, but in a small sample of introductory programming books available from Amazon [5, 7, 18, 20, 29] we find that this is actually representative of the kinds of problems students face when they first learn about loops. In the data collected from PEX4FUN website, we found several students trying multiple attempts (10+) to solve the problem before giving up. We have observed similar phenomenon in the data

[2] Pex4Fun http://pex4fun.com

[3] http://www.collegeboard.com/student/testing/ap/sub_compscia.html

collected from the introductory python programming course taught at MIT.

Figure 1 illustrates one of the challenges of grading and providing feedback for such programming problems. Part (a) of the figure shows a student's solution for array reversal taken from the Pex4Fun website. The implementation has two small errors: (i) `back++` should be `back--` and (ii) the while condition (`front > back`) should be (`front < back`), but this solution is really different from the reference implementation shown in Figure 1(b). Pointing out the errors in a student's solution requires detailed reasoning about the semantics of both programs. This makes grading very difficult, even for human graders; on the other hand, the alternative of grading students based on random testing of their code does not do justice to students who, like this student, are only a couple of typos away from a correct solution. Our tool is able to provide direct feedback about what is wrong with a given program, and give suggestions on how to fix it. For example, for this problem, the tool produces the following feedback:

```
1. Change the conditional (front > back) in line
10 to (front < back).
2. Change the increment back++ in line 13 to back--
```

One of the benefits of such direct feedback is that students get to understand the error in their code, instead of simply being told that they did it all wrong and being shown a (possibly quite different) correct solution. In order to maximize this benefit, it is useful to show corrections to students that are faithful to the solution strategy they were attempting. When there are many possible fixes to a program, a reasonable hypothesis is that the solution requiring the least number of fixes will be most faithful to the strategy that the student was pursuing. That may not always be the case, so our system also supports showing alternate ways of correcting the error. For example, Figure 2 shows an instance where multiple corrections are possible, and our tool can provide feedback about different corrections (two such feedback are shown in the figure).

```
1 using System;

3 public class Program {
4   public static int[] Puzzle(int[] b) {
5     for (int i=1; i<=b.Length/2; i++){
6       int temp = b[i];
7       b[i] = b[b.Length-i];
8       b[b.Length-i] = temp;
9     }
10    return b;
11  }
12 }
```

```
Fix 1: Decrement index i by 1 in lines 6 and 7.

Fix 2: Decrement index b.Length-i by 1 in lines 7
       and 8, Change loop initiation i to 0 in line 5.
```

Figure 2. Two possible fixes to fix the student's solution.

2.1 Workflow

In order to provide the level of feedback described above, the tool needs some information from the instructor. First, the tool needs to know what the problem is that the students are supposed to solve. The instructor provides this information by writing a reference implementation such as the one in Figure 1(b).

In addition to the reference implementation, the instructor also needs to describe to the tool what kinds of errors students might make. Our framework provides an error model language EML, which instructors can use to describe a set of correction rules that denote the potential corrections to errors that students might make. For example, in the two instances above, we observe that corrections often involve modifying an array index or a comparison operator, or replacing increments with decrements. The instructor can specify this information with the following three correction rules:

$$\begin{aligned} v[a] &\rightarrow v[a-1] \\ a_0 > a_1 &\rightarrow a_0 < a_1 \\ a{+}{+} &\rightarrow a{-}{-} \end{aligned}$$

The correction rule $v[a] \rightarrow v[a-1]$ states that an array access expression of the form $v[a]$ (lhs) in the program can be optionally replaced by an expression of the form $v[a-1]$ (rhs). These rules are too specific for this example, but as we will see in Section 3, they can be easily generalized to correct most of the student mistakes for this problem. In later experiments, we show how only a few tens of incorrect solutions can provide enough information to create an error model that can automatically provide feedback for hundreds of incorrect solutions. But in this section, we will only consider these rules to simplify the presentation.

The tool now needs to explore the space of all candidate programs based on these correction rules. We use constraint-based synthesis technology [12, 30, 33] to efficiently search over this large space of programs. Specifically, we use the SKETCH synthesizer that uses a sat-based algorithm to complete program sketches (programs with holes) so that they meet a given specification. To simplify the presentation, we use a simple imperative language IMP in place of C# to explain the details of our algorithm.

2.2 Solution Strategy

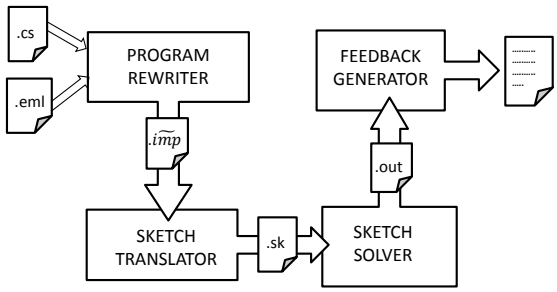

Figure 3. The architecture of our automated grading tool.

The architecture of our tool is shown in Figure 3. The solution strategy to find minimal corrections to a student's solution is based on a two-phase translation to the Sketch synthesis language. In the first phase, the `Program Rewriter` uses the correction rules to translate the solution into a language we call $\widetilde{\text{IMP}}$; this language provides us with a concise notation to describe sets of candidate programs, together with a cost model to reflect the number of corrections associated with each program in this set. In the second phase, this $\widetilde{\text{IMP}}$ program is translated into a sketch program by the `Sketch Translator`.

In the case of example in Figure 2, the `Program Rewriter` produces the $\widetilde{\text{IMP}}$ program shown in Figure 4 using the correction rules from Section 2.1. This program includes all the possible corrections induced by the correction rules in the model. The $\widetilde{\text{IMP}}$ language extends the imperative language IMP with expression choices, where the choices are denoted with squiggly brackets. Whenever there are multiple choices for an expression or a statement, the zero-cost

choice, the one that will leave the expression unchanged, is boxed. For example, the expression choice $\{\boxed{a_0}, a_1, \cdots, a_n\}$ denotes a choice between expressions $a_0, \cdots, a_n$ where a_0 denotes the zero-cost default choice.

For this simple program, the three correction rules induce a space of 32 different candidate programs. This candidate space is fairly small, but the number of candidate programs grow exponentially with the number of correction places in the program as well as with the number of correction choices in the rules. The error model that we use in our experiments induces a space of more than 10^{11} different programs for some of the benchmark problems. In order to search this large space efficiently, the program is translated to a sketch by the `Sketch Translator`.

SKETCH *(Background)* The SKETCH [30] synthesis system allows programmers to write programs while leaving fragments of it unspecified as *holes*; the contents of these holes are filled up automatically by the synthesizer such that the program conforms to a specification provided in terms of a reference implementation. The synthesizer uses the `CEGIS` algorithm [31] to efficiently compute the values for holes and uses bounded symbolic verification techniques for performing equivalence check of the two implementations.

2.3 Synthesizing Corrections with Sketch

The sketch generated for the example program is shown in Figure 5. A global integer variable `cost` is defined at the top that keeps track of the number of corrections performed to the original program. The SKETCH construct ?? denotes an unknown integer hole that can be assigned any constant integer value by the synthesizer. The expression choices in $\widetilde{\text{IMP}}$ are translated to functions in SKETCH that based on the unknown hole values return either the default expression or one of the other expression choices after incrementing the `cost` variable. For example, the `IndF1` function returns the input integer `i` if the hole value is 1 or otherwise returns the value `i-1` with the `cost` variable incremented. An assert statement (`assert cost <= k;`) is added towards the end of the sketch, where k is a constant parameter to the sketch that is initialized to 0. The synthesizer tries to find a solution that requires at most k modifications to the original program, and the parameter k is linearly incremented by 1 until the synthesizer finds a solution. The main idea behind the sketch encoding is that whenever the synthesizer tries to modify either an expression or a statement in the original program, the global variable `cost` is incremented by 1. By incrementing k linearly, the synthesizer is able to compute minimum cost corrections to a student's solution.

After the synthesizer finds a solution, the value of parameter k is used to assign the partial grade to the student's solution and the `Feedback Generator` uses the solution to the unknown integer holes in the sketch to compute the choices made by the synthesizer and generates the corresponding feedback. For this example, the tool generates the feedback shown in Figure 2 in less than 4 seconds.

The remainder of the paper describes each of these steps in more detail. We describe the syntax and semantics of our error model language EML in Section 3. Section 4 formalizes the translation of a student program to a sketch and describes the algorithm to find minimum corrections. We present the experimental evaluation of our tool on thousands of student solutions obtained from the PEX4FUN website in Section 5. We finally describe several work related to our technique in Section 6.

3. EML: Error Model Language

In this section, we describe the syntax and semantics of the error model language EML. An EML error model consists of a set of rewrite rules that captures the potential corrections for mistakes that

```
1  method int[] Puzzle(int[] b) {
2    for (int i=1; i<=b.Length/2; {i++,i--}){
3      int temp = b[{i,i-1}];
4      b[{i,i-1}] = b[{b.Length-i,b.Length-i-1}];
5      b[{b.Length-i,b.Length-i-1}] = temp;
6    }
7    return b;
8  }
9 }
```

Figure 4. The resulting $\widetilde{\text{IMP}}$ program after applying correction rules to program in Figure 2.

```
1  int cost = 0;
2  int[N] Puzzle(int[N] b){
3    for (int i=1; i<=N/2; IncF1(i)){
4      int temp = b[IndF1(i)];
5      b[IndF2(i)] = b[IndF3(b.Length-i)];
6      b[IndF4(b.Length-i)] = temp;
7    }
8    assert cost <= k;
9    return b;
10 }

12 int IndF1(int i){
13   if(??) return i;
14   else{ cost++; return i-1;}}
15 ...
16 void IncF1(ref int i){
17   if(??) i++;
18   else{ cost++; i--;}}
19 ...
```

Figure 5. The sketch generated for $\widetilde{\text{IMP}}$ program in Figure 4.

students might make in their solutions. We define the rewrite rules over a simple imperative language IMP, which is a small extension of Winskel's imperative language [36]. A rewrite rule transforms a program element in IMP to a set of weighted IMP program elements. This weighted set of IMP program elements is represented succinctly as an $\widetilde{\text{IMP}}$ program element, where $\widetilde{\text{IMP}}$ extends the IMP language with set-exprs (set of expressions) and set-stmts (set of statements). The weight associated with a program element in this set denotes the cost of performing the corresponding correction. An error model transforms an IMP program to an $\widetilde{\text{IMP}}$ program (representing a set of IMP programs) by recursively applying the rewrite rules. We show that this transformation is deterministic and is guaranteed to terminate on *well-formed* error models.

3.1 IMP and $\widetilde{\text{IMP}}$ languages

The syntax for the simple imperative language IMP is shown in Figure 6(a) and its semantics can be found in [36]. The syntax of $\widetilde{\text{IMP}}$ language is shown in Figure 6(b). The purpose of $\widetilde{\text{IMP}}$ language is to represent a large collection of IMP programs succinctly. The $\widetilde{\text{IMP}}$ language consists of set-expressions ($\tilde{a}$ and $\tilde{b}$) and set-statements ($\tilde{s}$) that represent a weighted set of corresponding IMP expressions and statements respectively. For example, the set expression $\{\boxed{n_0}, \cdots, n_k\}$ represents a weighted set of constant integers where n_0 denotes the default integer value associated with cost 0 and all other integer constants $(n_1, \cdots, n_k)$ are associated with cost 1. The sets of composite expressions are represented suc-

Arithmetic Expression a	:=	$n \mid v \mid a[a] \mid a_0\ op_a\ a_1$
	$\mid$	$f(a_0, \cdots, a_n)$
Arithmetic Operator op_a	:=	$+ \mid - \mid \times \mid /$
Boolean Expression b	:=	$\neg\, b \mid a_0\ op_c\ a_1 \mid b_0\ op_b\ b_1$
Comparison Operator op_c	:=	$== \mid < \mid > \mid \leq \mid \geq$
Boolean Operator op_b	:=	$\wedge \mid \vee$
Statement Expression s	:=	$v := a; \mid s_0; s_1 \mid$ `while` b `do` s
	$\mid$	`if` b `then` s_0 `else` s_1
	$\mid$	`return` a
Program p	:=	`method` $f(a_1, \cdots, a_n)\ s$

(a) IMP

Arithmetic set-expr $\tilde{a}$	:=	$a \mid \{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\}$
	$\mid$	$\tilde{a}[\tilde{a}] \mid \tilde{a}_0\ \widetilde{op}_a\ \tilde{a}_1$
	$\mid$	$\tilde{f}(\tilde{a}_0, \cdots, \tilde{a}_n)$
set-operator $\widetilde{op}_x$	:=	$op_a \mid \{\boxed{\widetilde{op}_{x_0}}, \cdots, \widetilde{op}_{x_n}\}$
Boolean set-expr $\tilde{b}$	:=	$b \mid \{\boxed{\tilde{b}_0}, \cdots, \tilde{b}_n\}$
	$\mid$	$\neg\, \tilde{b} \mid \tilde{a}_0\ \widetilde{op}_c\ \tilde{a}_1 \mid \tilde{b}_0\ \widetilde{op}_b\ \tilde{b}_1$
Statement set-stmt $\tilde{s}$	:=	$s \mid \{\boxed{\tilde{s}_0}, \cdots, \tilde{s}_n\}$
	$\mid$	$\tilde{v} := \tilde{a}; \mid \tilde{s}_0; \tilde{s}_1$
	$\mid$	`while` $\tilde{b}$ `do` $\tilde{s}$
	$\mid$	`if` $\tilde{b}$ `then` $\tilde{s}_0$ `else` $\tilde{s}_1$
	$\mid$	`return` $\tilde{a}$
Program $\tilde{p}$	:=	`method` $f(a_1, \cdots, a_n)\ \tilde{s}$

(b) $\widetilde{\text{IMP}}$

Figure 6. The syntax for (a) IMP and (b) $\widetilde{\text{IMP}}$ languages.

cinctly in terms of sets of their constituent sub-expressions. For example, the composite expression $\{\boxed{a_0}, a_0+1\}\{\boxed{<}, \leq, >, \geq, ==, \neq\}\{\boxed{a_1}, a_1+1, a_1-1\}$ represents 36 IMP expressions.

Each IMP program in the set of IMP programs represented by an $\widetilde{\text{IMP}}$ program is associated with a cost (weight) that encodes the number of modifications performed in the original program to obtain the transformed program. This cost allows the tool to search for corrections that require minimum number of modifications. The weighted set of IMP programs is defined using the $[\![\ ]\!]$ function shown in Figure 7. The $[\![\ ]\!]$ function on IMP expressions such as a returns a singleton set consisting of the corresponding expression associated with cost 0. On set-expressions of the form $\{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\}$, the function returns the union of the weighted set of IMP expressions corresponding to the default set-expression ($[\![\tilde{a}_0]\!]$) and the weighted set of expressions corresponding to other set-expressions ($\tilde{a}_1, \cdots, \tilde{a}_n$) in which each expression is associated with an additional cost of 1. On composite expressions, the function computes the weighted set recursively by taking the cross-product of weighted sets of its constituent sub-expressions and adding their corresponding costs. For example, the weighted set for composite expression $\tilde{x}[\tilde{y}]$ consists of an expression $x_i[y_j]$ associated with cost $c_{x_i} + c_{y_j}$ for each $(x_i, c_{x_i}) \in [\![\tilde{x}]\!]$ and $(y_j, c_{y_j}) \in [\![\tilde{y}]\!]$.

3.2 Syntax of EML

An EML error model consists of a set of correction rules that are used to transform an IMP program to an $\widetilde{\text{IMP}}$ program. A *correction rule* $\mathcal{C}$ is written as a rewrite rule $L \rightarrow R$, where L and R denote a *program element* in IMP and $\widetilde{\text{IMP}}$ respectively. A program element can either be a term, an expression, a statement, a method or the program itself. The left hand side (L) denotes an IMP program element that is pattern matched to be transformed to an $\widetilde{\text{IMP}}$ program element denoted by the right hand side (R). The left hand side of the rule can use free variables whereas the right hand side can only refer to the variables present in the left hand side. The language also supports a special $'$ (prime) operator that can be used to *tag* sub-expressions in R that are further transformed recursively using the error model.

The rules use a shorthand notation $?a$ (in the right hand side) to denote the set of all variables that are of the same type as the type of expression a and that are also in scope at the corresponding program location. Without this notation, the instructor would need to know all variables in the program beforehand to provide rewrite rules that involve replacing a variable.

Example 1. *The error model for the array-sort problem is shown in Figure 8. The* INDR *rewrite rule transforms the array access indices. The* INITR *rule transforms the right hand size of constant initializations. The* COMPR *rule transforms the operands and operator of the comparisons. The* INCR *rule transforms the increment statements. Note that these rewrite rules define the corrections that can be performed optionally; the zero cost (default) case of not correcting a program element is added automatically as described in Section 3.3.*

$$
\begin{aligned}
\text{INDR: } v[a] &\rightarrow v[\{a+1, a-1, ?a\}] \\
\text{INITR: } v = n &\rightarrow v = \{n+1, n-1, 0\} \\
\text{COMPR: } a_0\ op_c\ a_1 &\rightarrow \{a_0' - 1, 0, 1, ?a_0\}\ \widetilde{op}_c \\
&\qquad \{a_1' - 1, 0, 1, ?a_1\} \\
&\text{where } \widetilde{op}_c = \{<, >, \leq, \geq, ==, \neq\} \\
\text{INCR: } v{+}{+} &\rightarrow \{{+}{+}v, -{-}v, v{-}{-}\}
\end{aligned}
$$

Figure 8. The error model $\mathcal{E}$ for array sort problem.

Example 2. *The error model for the factorial problem is shown in Figure 9. The* FUNCR *rewrite rule transforms the method arguments in recursive calls. The* BASER *rule adds the base case for the factorial function to the method. The* INITR *rule transforms the right hand size of constant assignments and the* COMPR *rule transforms the comparisons in a similar way as in Example 1.*

Definition 1. *Well-formed Rewrite Rule : A rewrite rule $\mathcal{C} : L \rightarrow R$ is defined to be well-formed if all tagged sub-terms t' in R have a smaller size syntax tree than that of L.*

The rewrite rule $\mathcal{C}_1 : v[a] \rightarrow \{(v[a])' + 1\}$ is not a well-formed rewrite rule as the size of the tagged sub-term ($v[a]$) of R is the

$$
\begin{array}{rcl}
[\![a]\!] & = & \{(a,0)\} \\
[\![\{\boxed{\tilde{a}_0},\cdots,\tilde{a}_n\}]\!] & = & [\![\tilde{a}_0]\!] \cup \{(a,\ c+1) \mid (a,c) \in [\![\tilde{a}_i]\!], 0 < i \leq n\} \\
[\![\tilde{a}_0[\tilde{a}_1]]\!] & = & \{(a_0[a_1],\ c_0+c_1) \mid (a_0,c_0) \in [\![\tilde{a}_0]\!], (a_1,c_1) \in [\![\tilde{a}_1]\!]\} \\
[\![\tilde{a}_0\ \widetilde{op}_x\ \tilde{a}_1]\!] & = & \{(a_0\ op_x\ a_1,\ c_0+c_x+c_1) \mid (a_0,c_0) \in [\![\tilde{a}_0]\!], (op_x,c_x) \in [\![\widetilde{op}_x]\!], (a_1,c_1) \in [\![\tilde{a}_1]\!], x \in \{a,b,c\}\} \\
[\![\tilde{f}(\tilde{a}_1,\cdots,\tilde{a}_n)]\!] & = & \{(f(a_1,\cdots,a_n),\ c_f+c_1\cdots+c_n) \mid (f,c_f) \in [\![\tilde{f}]\!], (a_i,c_i) \in [\![\tilde{a}_i]\!]\} \\
[\![op_a]\!] & = & \{(op_a,0)\} \\
[\![\{\boxed{\widetilde{op}_{a_0}},\cdots,\widetilde{op}_{a_n}\}]\!] & = & [\![\widetilde{op}_{a_0}]\!] \cup \{(op_a,\ c+1) \mid (op_a,c) \in [\![\widetilde{op}_{a_i}]\!]\} \\
[\![b]\!] & = & \{(b,0)\} \\
[\![\{\boxed{\tilde{b}_0},\cdots,\tilde{b}_n\}]\!] & = & [\![\tilde{b}_0]\!] \cup \{(b,\ c+1) \mid (b,c) \in [\![\tilde{b}_i]\!], 0 < i \leq n\} \\
[\![s]\!] & = & \{(s,0)\} \\
[\![\{\boxed{\tilde{s}_0},\cdots,\tilde{s}_n\}]\!] & = & [\![\tilde{s}_0]\!] \cup \{(s,\ c+1) \mid (s,c) \in [\![\tilde{s}_i]\!], 0 < i \leq n\} \\
[\![\tilde{v} := \tilde{a}]\!] & = & \{(v := a,\ c_0+c_1) \mid (v,c_0) \in [\![\tilde{v}]\!], (a,c_1) \in [\![\tilde{a}]\!]\} \\
[\![\tilde{s}_0;\tilde{s}_1]\!] & = & \{(s_0;s_1,\ c_0+c_1) \mid (s_0,c_0) \in [\![\tilde{s}_0]\!], (s_1,c_1) \in [\![\tilde{s}_1]\!]\} \\
[\![\texttt{if}\ \tilde{b}\ \texttt{then}\ \tilde{s}_0\ \texttt{else}\ \tilde{s}_1]\!] & = & \{(\texttt{if}\ b\ \texttt{then}\ s_0\ \texttt{else}\ s_1,\ c_b+c_0+c_1) \mid (b,c_b) \in [\![\tilde{b}]\!], (s_0,c_0) \in [\![\tilde{s}_0]\!], (s_1,c_1) \in [\![\tilde{s}_1]\!]\} \\
[\![\texttt{while}\ \tilde{b}\ \texttt{do}\ \tilde{s}]\!] & = & \{(\texttt{while}\ b\ \texttt{do}\ s,\ c_b+c_s) \mid (b,c_b) \in [\![\tilde{b}]\!], (s,c_s) \in [\![\tilde{s}]\!]\} \\
[\![\texttt{return}\ \tilde{a}]\!] & = & \{(\texttt{return}\ a,\ c) \mid (a,c) \in [\![a]\!]\}
\end{array}
$$

Figure 7. The translation of a program in $\widetilde{\text{IMP}}$ to a set of programs in IMP, each associated with a cost c.

$$
\begin{array}{rcl}
\text{FUNCR:}\ f(a_1,\cdots,a_n) & \rightarrow & f(\tilde{a}_1,\cdots,\tilde{a}_n) \\
\text{where } \tilde{a}_i = \{a_i+1, a_i-1\} & & \\
\text{BASER:}\ \texttt{method fact}(a)\ s & \rightarrow & \texttt{method fact}(a)\ s_{base}; s \\
\text{where } s_{base} = \texttt{if}\ (a == 0)\ \texttt{return}\ 1; & & \\
\text{INITR:}\ v = n & \rightarrow & v = \{n+1, n-1, 1, 0\} \\
\text{COMPR:}\ a_0\ op_c\ a_1 & \rightarrow & \{a_0-1,0\}\ \widetilde{op}_c\ \{a_1-1,0\} \\
\text{where } \widetilde{op}_c = \{<,>,\leq,\geq,==,\neq\} & &
\end{array}
$$

Figure 9. The error model $\mathcal{E}$ for factorial problem.

same as that of the left hand side L. On the other hand, the rewrite rule $\mathcal{C}_2 : v[a] \rightarrow \{v'[a']+1\}$ is well-formed.

Definition 2. *Well-formed Error Model : An error model $\mathcal{E}$ is defined to be well-formed if all of its constituent rewrite rules $\mathcal{C}_i \in \mathcal{E}$ are well-formed.*

3.3 Transformation with EML

An error model $\mathcal{E}$ is syntactically translated to a function $\mathcal{T}_\mathcal{E}$ that transforms an IMP program to an $\widetilde{\text{IMP}}$ program as shown in Figure 10. The $\mathcal{T}_\mathcal{E}$ function first traverses the program element w in the default way, i.e. no transformation happens at this level of the syntax tree, and the method is called recursively on all of its top-level sub-terms t to obtain the transformed element $w_0 \in \widetilde{\text{IMP}}$. For each correction rule $\mathcal{C}_i : L_i \rightarrow R_i$ in the error model $\mathcal{E}$, the method contains a `Match` expression that matches the term w with the left hand side of the rule L_i (with appropriate unification of the free variables in L_i). If the match succeeds, it is transformed to a term $w_i \in \widetilde{\text{IMP}}$ as defined by the right hand side R_i of the rule after applying the $\mathcal{T}_\mathcal{E}$ method recursively on each one of its tagged sub-terms t'. Finally, the method returns the set of all transformed terms $\{\boxed{w_0},\cdots,w_n\}$.

$$
\begin{array}{l}
\mathcal{T}_\mathcal{E}(w : \text{IMP}) : \widetilde{\text{IMP}} = \\
\quad \texttt{let}\ w_0 = w[t \rightarrow \mathcal{T}_\mathcal{E}(t)]\ \texttt{in} \\
\quad\quad \cdots\cdots \\
\quad \texttt{let}\ w_i = \texttt{Match}\ w\ \texttt{with} \\
\quad\quad\quad L_i\ \rightarrow\ R_i[t'\ \rightarrow\ \mathcal{T}_\mathcal{E}(t)]\ \texttt{in} \\
\quad\quad \cdots \\
\quad \{\boxed{w_0},\cdots,w_n\}
\end{array}
$$

Figure 10. The syntactic translation of an error model $\mathcal{E}$ to $\mathcal{T}_\mathcal{E}$ function.

Example 3. *Consider an error model $\mathcal{E}_1$ consisting of the following three correction rules:*

$$
\begin{array}{rcl}
\mathcal{C}_1 :\ v[a] & \rightarrow & v[\{a-1, a+1\}] \\
\mathcal{C}_2 :\ a_0\ op_c\ a_1 & \rightarrow & \{a_0'-1,0\}\ op_c\ \{a_1'-1,0\} \\
\mathcal{C}_3 :\ v[a] & \rightarrow & ?v[a]
\end{array}
$$

The transformation function $\mathcal{T}_{\mathcal{E}_1}$ for the error model $\mathcal{E}_1$ is shown in Figure 11. The recursive steps of application of $\mathcal{T}_{\mathcal{E}_1}$ function on expression $(x[i] < y[j])$ are shown in Figure 12.

This example illustrates two interesting features of the transformation function:

- Nested Transformations : Once a rewrite rule $L \rightarrow R$ is applied to transform a program element matching L to R, the instructor may want to apply another rewrite rule on only a few sub-terms of R. For example, she may want to avoid transforming the sub-

$$\mathcal{T}(x[i] < y[j]) \equiv \{\boxed{\mathcal{T}(x[i]) < \mathcal{T}(y[j])}, \{\mathcal{T}(x[i]) - 1, 0\} < \{\mathcal{T}(y[j]) - 1, 0\}\}$$

$$\mathcal{T}(x[i]) \equiv \{\boxed{\mathcal{T}(x)[\mathcal{T}(i)]}, x[\{i+1, i-1\}], y[i]\}$$

$$\mathcal{T}(y[j]) \equiv \{\boxed{\mathcal{T}(y)[\mathcal{T}(j)]}, y[\{j+1, j-1\}], x[j]\}$$

$$\mathcal{T}(x) \equiv \{\boxed{x}\} \qquad \mathcal{T}(i) \equiv \{\boxed{i}\} \qquad \mathcal{T}(y) \equiv \{\boxed{y}\} \qquad \mathcal{T}(j) \equiv \{\boxed{j}\}$$

Therefore, after substitution the result is:

$$\mathcal{T}(x[i] < y[j]) \equiv \{\{\boxed{\boxed{x}[\boxed{i}]}, x[\{i+1, i-1\}], y[i]\} < \{\boxed{\boxed{y}[\boxed{j}]}, y[\{j+1, j-1\}], x[j]\}\},$$

$$\{\{\boxed{\boxed{x}[\boxed{i}]}, x[\{i+1, i-1\}], y[i]\} - 1, 0\} < \{\{\boxed{\boxed{y}[\boxed{j}]}, y[\{j+1, j-1\}], x[j]\} - 1, 0\}\}$$

Figure 12. Application of $\mathcal{T}_{\mathcal{E}_1}$ (abbreviated $\mathcal{T}$) on expression $(x[i] < y[j])$.

$$\begin{aligned}
&\mathcal{T}_{\mathcal{E}_1}(w : \text{IMP}) : \widetilde{\text{IMP}} \quad = \\
&\texttt{let } w_0 = w[t \rightarrow \mathcal{T}_{\mathcal{E}_1}(t)] \texttt{ in} \\
&\texttt{let } w_1 = \texttt{Match } w \texttt{ with} \\
&\qquad\qquad v[a] \rightarrow v[\{a+1, a-1\}] \texttt{ in} \\
&\texttt{let } w_2 = \texttt{Match } w \texttt{ with} \\
&\qquad\qquad a_0\ op_c\ a_1 \rightarrow \{\mathcal{T}_{\mathcal{E}_1}(a_0) - 1, 0\}\ op_c \\
&\qquad\qquad\qquad\qquad \{\mathcal{T}_{\mathcal{E}_1}(a_1) - 1, 0\} \texttt{ in} \\
&\{\boxed{w_0}, w_1, w_2\}
\end{aligned}$$

Figure 11. The $\mathcal{T}_{\mathcal{E}_1}$ method for error model $\mathcal{E}_1$.

terms which have already been transformed by some other correction rule. The EML language facilitates making such distinction between the sub-terms for performing nested corrections using the $\prime$ (prime) operator. Only the sub-terms in R that are tagged with the prime operator are visited for applying further transformations (using the $\mathcal{T}_{\mathcal{E}}$ method recursively on its tagged sub-terms t'), whereas the remaining non-tagged sub-terms are not transformed any further. After applying the rewrite rule $\mathcal{C}_2$ in the example, the sub-terms $x[i]$ and $y[j]$ are further transformed by applying rewrite rules $\mathcal{C}_1$ and $\mathcal{C}_3$.

- Ambiguous Transformations : While transforming a program using an error model, it may happen that there are multiple rewrite rules that pattern match the program element w. After applying rewrite rule $\mathcal{C}_2$ in the example, there are two rewrite rules $\mathcal{C}_1$ and $\mathcal{C}_3$ that pattern match the terms $x[i]$ and $y[j]$. After applying one of these rules ($\mathcal{C}_1$ or $\mathcal{C}_3$) to an expression $v[a]$, we cannot apply the other rule to the transformed expression. In such ambiguous cases, the $\mathcal{T}_{\mathcal{E}}$ function creates a separate copy of the transformed program element (w_i) for each ambiguous choice and then performs the set union of all such elements to obtain the transformed program element. This semantics of handling ambiguity of rewrite rules also matches naturally with the intent of the instructor. If the instructor wanted to perform both transformations together on array accesses, she could have provided a combined rewrite rule such as $v[a] \rightarrow ?v[\{a+1, a-1\}]$.

Theorem 1. *Given a well-formed error model $\mathcal{E}$, the transformation function $\mathcal{T}_{\mathcal{E}}$ always terminates.*

Proof. From the definition of well-formed error model, each of its constituent rewrite rule is also well-formed. Hence, each application of a rewrite rule is guaranteed to reduce the size of the syntax tree of terms that are required to be visited further for transformation by $\mathcal{T}_{\mathcal{E}}$. Therefore, the $\mathcal{T}_{\mathcal{E}}$ function terminates in a finite number of steps. □

4. Constraint-based Solving of $\widetilde{\text{IMP}}$ programs

In the previous section, we saw how the tool transforms an IMP program to an $\widetilde{\text{IMP}}$ program using an error model. We now present the translation of $\widetilde{\text{IMP}}$ programs into SKETCH programs [30]. In this section, we use the `if` expressions for formalizing the translation but in practice, these expressions are encapsulated into functions that makes the SKETCH output amenable for providing feedback corresponding to the corrections. We also present the algorithm to solve the sketch program to compute a minimum cost IMP program from the set of programs represented by the $\widetilde{\text{IMP}}$ program that is semantically equivalent to the instructor's reference implementation.

4.1 Translation of $\widetilde{\text{IMP}}$ programs to SKETCH

We show the translation (Φ) of some of the interesting $\widetilde{\text{IMP}}$ constructs to SKETCH expressions in Figure 13. The SKETCH construct ?? (called *hole*) is a placeholder for a constant value, which is filled up by the SKETCH synthesizer while solving the constraints to satisfy the given specification. The translation defines a global integer variable `cost` that is used to compute the total number of modifications performed to the original program to obtain the transformed program.

The singleton sets consisting of an IMP expression such as $\{a\}$ are translated simply to the corresponding expression itself. A set-expression of the form $\{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\}$ is translated recursively to the `if` expression `:if` $(??)$ $\Phi(\tilde{a}_0)$ `else` $\Phi(\{\tilde{a}_1, \cdots, \tilde{a}_n\})$, which means that the synthesizer can optionally select the default set-expression $\Phi(\tilde{a}_0)$ (by choosing ?? to be `true`) or select one of the other choices $(\tilde{a}_1, \cdots, \tilde{a}_n)$. The set-expressions of the form $\{\tilde{a}_0, \cdots, \tilde{a}_n\}$ are similarly translated but with an additional statement for incrementing the `cost` variable if the synthesizer selects the non-default choice $\tilde{a}_0$.

The translation rules for the assignment statements ($\tilde{a}_0 := \tilde{a}_1$) results in `if` expressions on both left and right sides of the assignment. The `if` expression choices occurring on the left hand side are desugared to individual assignments. For example, the left

$$
\begin{array}{rcl}
\Phi(\{a\}) & = & a \\
\Phi(\{\boxed{\tilde{a}_0},\cdots,\tilde{a}_n\}) & = & \texttt{if}\ (??)\ \Phi(\tilde{a}_0)\ \texttt{else}\ \Phi(\{\tilde{a}_1,\cdots,\tilde{a}_n\}) \\
\Phi(\{\tilde{a}_0,\cdots,\tilde{a}_n\}) & = & \texttt{if}\ (??)\ \{\texttt{cost} = \texttt{cost}+1; \Phi(\tilde{a}_0)\} \\
 & & \texttt{else}\ \Phi(\{\tilde{a}_1,\cdots,\tilde{a}_n\}) \\
\Phi(\tilde{a}_0[\tilde{a}_1]) & = & \Phi(\tilde{a}_0)[\Phi(\tilde{a}_1)] \\
\Phi(\{f(\tilde{a}_1,\cdots,\tilde{a}_n)\}) & = & f(\Phi(\tilde{a}_1),\cdots,\Phi(\tilde{a}_n)) \\
\Phi\{\boxed{\tilde{f}_0},\cdots,\tilde{f}_n\}(\tilde{a}_1, & = & \texttt{if}\ (??)\ \Phi(\tilde{f}_0(\tilde{a}_1,\cdots,\tilde{a}_n)) \\
\tilde{a}_2,\cdots,\tilde{a}_n) & & \texttt{else}\ \Phi(\{\tilde{f}_1,\cdots,\tilde{f}_n\}(\tilde{a}_1,\cdots,\tilde{a}_n)) \\
\Phi\{\tilde{f}_0,\cdots,\tilde{f}_n\}(\tilde{a}_1, & = & \texttt{if}\ (??)\ \{\texttt{cost} = \texttt{cost}+1; \\
\tilde{a}_2\cdots,\tilde{a}_n) & & \qquad \Phi(\tilde{f}_0(\tilde{a}_1,\cdots,\tilde{a}_n))\} \\
 & & \texttt{else}\ \Phi(\{\tilde{f}_1,\cdots,\tilde{f}_n\}(\tilde{a}_1,\cdots,\tilde{a}_n)) \\
\Phi(\tilde{a}_0 := \tilde{a}_1) & = & \Phi(\tilde{a}_0) := \Phi(\tilde{a}_1) \\
\Phi(\tilde{s}_0; \tilde{s}_1) & = & \Phi(\tilde{s}_0); \Phi(\tilde{s}_1) \\
\Phi(\texttt{if}\ \tilde{b}\ \texttt{then}\ \tilde{s}_0\ \texttt{else}\ \tilde{s}_1) & = & \texttt{if}\ (\Phi(\tilde{b}))\ \{\Phi(\tilde{s}_0)\}\ \texttt{else}\ \{\Phi(\tilde{s}_1)\} \\
\Phi(\texttt{while}\ \tilde{b}\ \texttt{do}\ \tilde{s}) & = & \texttt{while}\ (\Phi(\tilde{b}))\ \{\Phi(\tilde{s})\} \\
\Phi(\texttt{return}\ \tilde{a}) & = & \texttt{return}\ \Phi(\tilde{a})
\end{array}
$$

Figure 13. Some of the interesting translation rules for converting an $\widetilde{\text{IMP}}$ program to a SKETCH program.

```
1 using System;
2 public class Program {
3   public static int Factorial(int x) {
4     int col = x;
5     for(int i = x-1 ; i >0; i--)
6       col*= i;
7     return col;
8   }}
```

Figure 14. A student solution for the factorial function with two possible fixes.

hand side expression `if` $(??)$ x `else` $y := 10$ is desugared to `if` $(??)$ $x := 10$ `else` $y := 10$. The infix operators in $\widetilde{\text{IMP}}$ are first translated to function calls and then are translated to sketch using the translation for set-function expressions. The remaining $\widetilde{\text{IMP}}$ expressions are similarly translated recursively as shown in the figure.

4.2 Computing the least-cost correct modification

Once we have translated an $\widetilde{\text{IMP}}$ program to a SKETCH program, the synthesizer now needs to solve these constraints to find a transformed program with the least `cost` such that it is semantically equivalent to the instructor's reference implementation. The tool first queries the synthesizer to find a solution with an added assertion (`assert cost` $==0$), i.e. the case in which the student program was correct in the first place and did not require any modification. If the synthesizer can find a solution, the tool exits the algorithm. Otherwise, the synthesizer states that the constraints are unsatisfiable. The tool then increments the `cost` variable by 1 and queries the solver for a solution with an assertion (`assert cost` $==1$), corresponding to the case in which the student program requires a single modification to make it semantically equivalent to the teacher's program. In this manner, the tool repeatedly queries the synthesizer with increasing values of `cost` until the constraint is satisfiable. We also put a bound on the maximum value of `cost` as k, such that if we can not find a solution for `assert cost` $== k$ we report back that the student solution can not be fixed using less than k modifications. We set this bound to $k = 5$ for our case studies, which was adequate for all of our benchmark examples. This algorithm calls the synthesizer $O(k)$ times, but we can easily adapt the algorithm to perform a binary search on k instead of this linear search. In the case of binary search, we would need to change the assertions on `cost` to the form (`assert cost` $\leq k$). Since we kept $k = 5$ for our experiments, we did not implement the binary search algorithm as k was small enough for linear search to scale just as well.

4.3 Computing Alternate Solutions

It might happen that in some cases the solution computed by our tool does not correspond to the way in which the student was planning to solve the problem. Consider a student solution to the factorial problem shown in Figure 14 that outputs 0 instead of 1 on the input $x = 0$. With the error model consisting of adding the base case, the tool finds a fix to add the base case for $x = 0$. With the error model consisting of initialization correction, the tool finds a solution where col is initialized to 1 and the for loop iterator i is initialized to x. Both the fixes look perfectly viable and it is hard to prefer one over the other. To cater such cases, our tool provides an option to ask for an alternate solution if the suggested fix is not in correspondence with the student's intent. The tool asserts an additional assertion that corresponds to the negation of all unknown hole choices from the previous solution, so that it can now search for a different solution.

4.4 Mapping SKETCH solution to generate feedback

Each transformation rule in the error model $\mathcal{E}$ is associated with a feedback message, e.g. the array index transformation from array-reverse error model (INDR)$v[a] \rightarrow v[a+1]$ is associated with a message "Increment the index a by 1 in line l" and the base case transformation (BASER) from factorial error model is associated with the message "Add missing base case of returning 1 on $x = 0$". After the SKETCH synthesizer finds a solution to the constraints, the tool maps back the values of unknown integer holes to their corresponding expression choices. These expression choices are then mapped to natural language feedback using the messages associated with the corresponding correction rules, together with the line numbers from the original program.

5. Implementation and Experiments

We have implemented the transformation and synthesis algorithms of our framework in a tool called `AutoGrader`. The tool consists of four main modules: i) Transformation module, ii) Translation module, iii) Solving module, and iv) Feedback Generation module as shown in Figure 3. The Transformation module parses and transforms a student solution (written in C#) to a program in $\widetilde{\text{C\#}}$ language ($\widetilde{\text{IMP}}$ variant of C#). We use Microsoft's Roslyn compiler framework[4] to perform the transformations on C# programs based on a set of correction rules that are encoded as a set of abstract syntax tree rewrite rules. The Translation module translates the $\widetilde{\text{C\#}}$ program into a SKETCH program, and then the Solving module solves the sketch program to compute minimal modifications in the original program such that it becomes semantically equivalent to the reference implementation. Finally, the Feedback Generation module maps the computed modifications in the original program to the corresponding explanations in natural language for generating the

[4] Microsoft Roslyn Framework http://msdn.microsoft.com/en-us/roslyn

feedback. We now describe our experience with the AutoGrader tool on several benchmark problems.

5.1 Benchmarks

We collected logs of 5 loops-over-arrays problems: array-reverse, palindrome, is array sorted, array max and array sort, as well as a recursion problem (factorial) from the PEX4FUN website. Our selection criteria for selecting these problems was to select the problems for which there was a clear specification. Typical puzzle problems on PEX4FUN website did not satisfy the criteria as there is no specification for them. In addition to these problems, we added three problems[5] that were slight variants of problems from AP high school computer science examinations. These problems include : (i) `Stock Market-I` – a program to check if there exists at least 3 pairs of consecutive elements in the stock prices array that differ by more than a given value, (ii) `stockMarket-II` – a program to compare the difference of `max` and `min` values of a given sub-range of stock prices array with a given value, and (iii) `Friday Restaurant Rush` – a variant of the maximum contiguous subsequence sum problem.

5.2 Evaluation Metric

We obtained PEX4FUN logs containing $10,245$ total attempts for these problems from which we constructed our test set by removing some attempts that fell into one of the following categories (as shown in Figure 15):

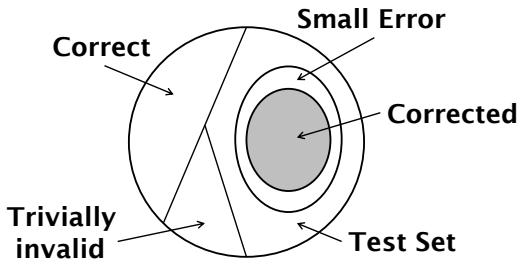

Figure 15. The breakdown of attempts for benchmark problems.

- **Correct:** attempts that were already correct and did not require any corrections.
- **Trivially invalid:** attempts that can be invalidated trivially.
  - **Compilation errors:** attempts with compilation errors.
  - **Library calls:** attempts that used library methods such as `Array.Sort`, `Array.Reverse` to directly solve the problem.
  - **Lists:** attempts that first converted arrays to generic lists and then used list functions for solving directly.
  - **No loops:** attempts that did not contain a loop for the loops-over-array problems.
- **Test Set:** attempts with non-trivial errors.
  - **Big error:** fixing such attempts would require completely rewriting the code.
  - **Small error:** attempts solving the correct problem.
  - **Corrected:** subset of small error attempts that were corrected by `AutoGrader`.

The detailed breakdown of the number of examples for each benchmark problem is shown in Table 1. We evaluate our tool on the success metric of `Effectiveness` (`Eff`) defined as $\texttt{Eff} = \frac{\texttt{Corrected}}{\texttt{SmallError}}$. The `Effectiveness` metric is important because it measures what fraction of correctable attempts our tool can correct. We also present the `Fixable Fraction` (`FF`) metric defined as $\texttt{FF} = \frac{\texttt{SmallError}}{\texttt{TestSet}}$ that denotes the fraction of problems that are amenable to be fixed with small local fixes. The `FF` measure is sensitive to the quality of the collected data. The data collected from the PEX4FUN website has lower `FF` measure (and larger `Big error` set) because PEX4FUN users often submit completely different solutions just to check how the system works. The `Eff` and `FF` measures for the benchmarks problems is shown in Table 2. The experiments were performed on an Intel Core 2 Quad 3.0GHz CPU machine with 8GB of RAM.

On average, `AutoGrader` was able to correct about 73% of small errors (`Eff`) and 31% (`Eff` $\times$ `FF`) of all incorrect attempts in the `TestSet`. Some examples of the small error cases that `AutoGrader` could not fix include: 1) swapping array elements `a[i]` and `a[j]` using the statements `a[i]=a[j]; a[j]=a[i];`, 2) using `i` instead of `a[i]`, 3) using `(a[i]>10)` instead of `(a[i]-a[i-1])>10` in a comparison etc. In principle, some of these mistakes can be handled by making our error models more specific with additional correction rules, but we started getting diminishing returns as can be seen in Figure 16(a).

Benchmark	Eff	FF	Time(s)	EC2 ($)
Array reverse	83.3%	54.5%	2.69	2.98
String palindrome	74.4%	28%	3.52	3.91
Array maximum	68.7%	31.7%	7.47	8.30
Is Array increasing	74.5%	30.5%	3.56	3.96
Array sort	47.3%	49.7%	32.46	36.07
Factorial	57.1%	30.4%	4.99	5.54
StockMarket-I	77.7%	100%	3.19	3.54
StockMarket-II	73.7%	79.2%	19.61	21.79
Friday rush	80.4%	77.3%	4.66	5.18

Table 2. The `Effectiveness` (`Eff`) and the `Fixable Fraction` (`FF`) ratios, the average time taken per problem and the EC2 cluster price of grading each problem for 100,000 submissions.

5.3 Repetitive mistakes

We first investigate our hypothesis that students make similar mistakes when solving a given problem. For each benchmark `TestSet`, we selected the first incorrect solution that can be fixed using our correction language and then we added the corresponding correction rule to the error model. We then re-ran the benchmarks to get another (smaller) set of failing solutions. We kept repeating this step of selecting the first incorrect solution and enhancing the error model until we achieved the maximum possible coverage of fixing the incorrect solutions.

The results of this experiment are shown in Figure 16(a). The horizontal axis shows different error models $\mathcal{E}$ in monotonically increasing order of number of correction rules, such that $\mathcal{E}_i \subset \mathcal{E}_{j \forall i<j}$. The vertical axis shows the number of incorrect solutions that were fixed using these error models. For the array reverse problem, the error models $\mathcal{E}_1 = \{\text{INITR}\}$ fixed 21 solutions, $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\text{COMPR}\}$ fixed 107 solutions, $\mathcal{E}_3 = \mathcal{E}_2 \cup \{\text{INDR}\}$ fixed 235 solutions, $\mathcal{E}_4 = \mathcal{E}_3 \cup \{\text{RETR}\}$ fixed 252 solutions and $\mathcal{E}_5 = \mathcal{E}_4 \cup \{\text{INCR}\}$ fixed 254 solutions. We had to investigate only 5 incorrect solutions and around 10-15 out-of-scope solutions to correct around 250 incorrect solutions. We obtained similar results for other benchmark problems as well, as shown in the figure. These results support our hypothesis that students do make similar mistakes while solving a given problem, and it is therefore possible

[5] http://www.pexforfun.com/learnbeginningprogramming

(a) (b) (c)

Figure 16. (a)The fraction of incorrect solutions fixed with varying error models $\mathcal{E}$, (b) performance of `AutoGrader` with varying complexity of error models, and (c) performance of array reverse error model on other loop-over-arrays problems.

Benchmark	Total	Correct	Trivially Invalid				TestSet		
			Comp. Err.	Library Calls	Lists	No Loops	Big Errors	Small Errors	Corrected
Array reverse	2547	288	1020	245	153	281	255	305	254
String palindrome	2130	134	861	147	12	669	221	86	64
Array maximum	2402	466	733	41	13	837	213	99	68
Is Array increasing	555	36	208	10	6	128	116	51	38
Array sort	1291	42	555	213	85	247	75	74	35
Factorial	1093	66	210	0	0	587	160	70	40
Stock Market-I	52	19	11	0	0	0	0	22	16
Stock Market-II	51	19	8	0	0	0	5	19	14
Friday Rush	124	20	38	0	0	0	15	51	41

Table 1. The breakdown of benchmark logs obtained from the PEX4FUN website for loops-over-arrays and AP exam problems.

to correct a large number of incorrect attempts by investigating only a handful of incorrect solutions.

5.4 Performance vs Error Model Complexity

We next compare how the performance of `AutoGrader` tool varies with change in complexity of error models in Figure 16(b). We hypothesize that the tool will take more time to compute fixes with increase in the complexity of error models due to the increase in size of the search space of possible corrections. The results in the figure, however, show that the running time does not increase monotonically rather remains almost constant with the increase in model complexity. A possible reason for this phenomenon is that even though the search space of corrections with a more complex error model becomes larger, there are also newer ways available for the synthesizer to correct problems that were not available before with a less complex error model.

5.5 Generalization of Error Models

We next investigate the hypothesis that error models generalize well across different problems of the same category. To perform this experiment, we took the error model for the array reverse problem and used it as the error model for other problems in the loops-over-arrays category. The results of this experiment are shown in Figure 16(c). As can be seen from the figure, the error model generalizes well for some problems and not so well for the others. But even for those problems where the model does not correct many mistakes, the instructors can use it as a good starting point and then specialize them by adding problem specific correction rules to achieve higher correction coverage.

5.6 Number of Corrections

The number of problems that require different number of corrections is shown in Figure 17 (with logarithmic axis). The figure shows that even though majority of solutions required only 1 correction, there are significant number of problems that require multiple corrections and in some cases even up to 5 corrections were required. This shows that a tool for correcting mistakes, like ours, needs to be able to perform multiple coordinated fixes. For example, consider a student attempt for the stockMarket-I problem shown in Figure 18. The solution requires 4 corrections namely: `sPrices.Length` to `sPrices.Length-1`, `(delta > 10)` to `(delta >= 10)`, `(-delta > 10)` to `(-delta >= 10)`, and `(count >= 3)` to `(count < 3)`.

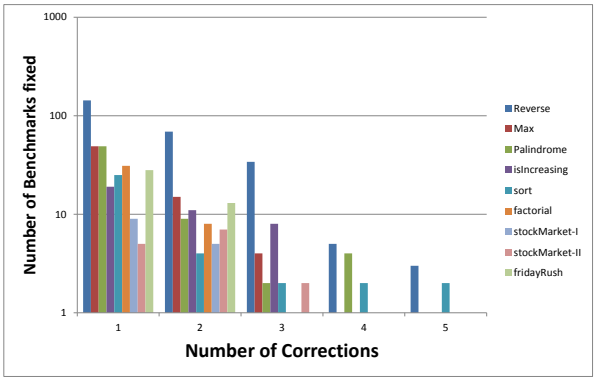

Figure 17. The number of incorrect solutions that require different number of modifications.

5.7 Correction Times

The average times for correcting each benchmark problem is shown in Table 2. Most of the benchmark problems take less than 10 seconds to find the corrections. The table also shows the column titled `EC2` that shows the price in `USD` that we would have to pay if

```
1 using System;
2 public class Program {
3     public static bool Puzzle(int[] sPrices) {
4       int count = 0;
5       for (int i = 0; i < sPrices.Length; i++) {
6         int delta = sPrices[i] - sPrices[i+1];
7         if (delta > 10 || -delta > 10)
8           count++;
9       }
10      if (count >= 3)
11        return true;
12      return false;
13    }
14 }
```

Figure 18. A student solution for stockMarket-I problem that requires 4 changes at places shown in the figure.

we were use to use the AutoGrader tool to grade 100,000 student assignments for each one of these problems. A few tens of dollars for grading 100,000 assignments is quite reasonable compared to the cost of hiring TAs or graders.

6. Related Work

In this section, we describe several related work to our technique from the areas of automated programming tutors, automated program repair, fault localization, automated debugging and program synthesis.

6.1 AI based programming tutors

There has been a lot of work done in the AI community for building automated tutors for helping novice programmers learn programming by providing feedback about semantic errors. These tutoring systems can be categorized in two major classes: (i) systems that match student's attempt with teacher's solution using some intermediate representation of the implementations and (ii) systems that try to infer the student intention (goals and plans) first and then match these intentions with that of the teacher.

Code-based matching approaches: LAURA [1] converts teacher's solution and student's attempt into a graph based representation. It then compares them by heuristically applying program transformations and reports mismatches as potential bugs. This approach works well if the teacher's algorithm is exactly the same as that of the student, but students typically solve the same problem using many different algorithms as we have observed in our dataset as well. TALUS [27] matches a student's attempt with a collection of teacher's algorithms. It first tries to recognize the algorithm used in the student's attempt and then compares the attempt with the recognized algorithm using symbolic evaluation on a set of cases. The top-level expressions in the student's attempt are replaced tentatively with expressions in the algorithm for generating correction feedback. The problem with this approach is that the enumeration of all possible algorithms (with its variants) for covering all corrections is very large and tedious on part of the teacher.

Intention-based matching approaches: LISP tutor [8] creates a model of the student goals and updates it dynamically as the student makes edits. For incorrect edits, it provides suggestions based on the teacher's model of inferred student intentions. The drawback of this approach is that it forces students to write code in a certain pre-defined structure and limits their freedom. MENO-II [32] parses student programs into a deep syntax tree whose nodes are annotated with plan tags. This annotated tree is then matched with the plans obtained from teacher's solution (provided in a high level language PDL). PROUST [22], on the other hand, uses a knowledge base of goals and their corresponding plans for implementing them for each programming problem. It first tries to find correspondence of these plans in the student's code and then performs matching to find discrepancies. CHIRON [28] is its improved version in which the goals and plans in the knowledge base are organized in a hierarchical manner based on their generality and uses machine learning techniques for plan identification in the student code. These approaches require teacher to provide all possible plans a student can use to solve the goals of a given problem and do not perform well if the student's attempt uses a plan not present in the knowledge base.

Our approach performs semantic equivalence of student's attempt and teacher's solution based on exhaustive bounded symbolic verification techniques and makes no assumptions on the algorithms or plans that students can use for solving the problem. Moreover, our approach is modular with respect to error models, as the teacher only needs to provide local correction rules in a declarative manner and their complex interactions are handled by the solver itself.

6.2 Automated Program Repair

Könighofer et. al. [25] recently presented an approach for automated error localization and correction of imperative programs. They use model-based diagnosis to localize components that need to be replaced and then use a template-based approach for providing corrections using SMT reasoning. Their fault model only considers the right hand side (RHS) of assignment statements as replaceable components. The approaches in [21, 34] frame the problem of program repair as a game between an environment that provides the inputs and a system that provides correct values for the buggy expressions such that the specification is satisfied. It isn't surprising that these approaches only support simple corrections (e.g. correcting RHS side of expressions) in the fault model as they aim to repair large programs with arbitrary errors. In our setting, we exploit the fact that beginner student programs are small and we have access to the dataset of previous student mistakes that we can use to construct a *concise and precise* fault model. This enables us to model more sophisticated transformations such as introducing new program statements, replacing LHS of assignments etc. in our error model. Our approach also supports minimal cost changes to student's programs where each error in the model is associated with a certain cost, unlike the earlier mentioned approaches.

Mutation-based program repair [6] performs mutations repeatedly to statements in a buggy program in order of their suspiciousness until the program becomes correct. The problem with this approach is that there is a large state space of mutants (approximately of the order of 10^{11} in our case studies) and explicitly enumerating them is infeasible. Our approach uses a symbolic search for exploring correct solutions over this large set of mutants. There are also some genetic programming approaches that exploit redundancy present in some other part of the code for fixing faults [2, 9]. These techniques are not applicable in our setting as we do not have such redundancy present in small introductory programming problems.

6.3 Automated Debugging and Fault localization

Techniques like Delta Debugging [37] and QuickXplain [24] aim to simplify a failing test case to a minimal test case that still exhibits the same failure. Our approach can be complemented with these techniques to restrict the application of rewrite rules to certain failing parts of the program only. There are many algorithms for fault localization [3, 10] that use the difference between faulty and successful executions of the system to identify potential faulty locations. Jose et. al. [23] recently suggested an approach that uses

a MAX-SAT solver to satisfy maximum number of clauses in a formula obtained from a failing test case to compute potential error locations. These approaches, however, only localize faults for a single failing test case and the suggested error location might not be the desired error location, since we are looking for common error locations that cause failure of multiple test cases. Moreover, these techniques provide only a limited set of suggestions (if any) for repairing these faults.

6.4 Program Synthesis

Program synthesis has been used recently for many applications such as synthesis of efficient low-level code [26, 31], inference of efficient synchronization in concurrent programs [35], bit-vector and graph algorithms [13, 19], snippets of excel macros [15], relational data structures [16, 17] and angelic programming [4]. The SKETCH tool [30, 31] takes a partial program and a reference implementation as input and uses constraint-based reasoning to synthesize a complete program that is equivalent to the reference implementation. In general cases, the template of the desired program as well as the reference specification is unknown and puts an additional burden on the users to provide them; in our case we use the student's solution as the template program and teacher's solution as the reference implementation. A recent work by Gulwani et. al. [14] also uses program synthesis techniques for automatically synthesizing solutions to ruler/compass based geometry construction problems. Their focus is primarily on finding a solution to a given geometry problem whereas we aim to grade and provide feedback on a given programming exercise solution. A comprehensive survey on various program synthesis techniques and different intention mechanisms for specification can be found in [11].

7. Conclusions

In this paper, we presented a new technique of automatically grading introductory programming assignments that can complement manual and test-cases based grading. The technique uses an error model describing the potential corrections and constraint-based synthesis to compute minimal corrections to student's incorrect solutions. We have evaluated our technique on a large set of benchmarks and it can correct 73% of fixable fraction of incorrect solutions with non-trivial errors. We believe this technique can provide a basis for providing automated feedback to hundreds of thousands of students learning from online introductory programming courses that are being planned to be taught by MITx and Udacity.

References

[1] A. Adam and J.-P. H. Laurent. LAURA, A System to Debug Student Programs. *Artif. Intell.*, 15(1-2):75–122, 1980.

[2] A. Arcuri. On the automation of fixing software bugs. In *ICSE Companion*, 2008.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

[4] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[5] M. Dawson. *Python Programming for the Absolute Beginner, 3rd Edition*. Course Technology PTR, 2010.

[6] V. Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, 2010.

[7] M. E. Farrell. *Computer Programming for Teens*. Course Technology PTR, 2007.

[8] R. G. Farrell, J. R. Anderson, and B. J. Reiser. An interactive computer-based tutor for lisp. In *AAAI*, 1984.

[9] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *GECCO*, 2009.

[10] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[11] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[12] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[13] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[14] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[15] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *CACM*, 2012. To Appear.

[16] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, 2011.

[17] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[18] M. Heidenry. *Java Programming for High School Students*. Reedy Press, 2009.

[19] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.

[20] Jerry Lee Ford Jr. *Programming for the Absolute Beginner*. Course Technology PTR, 2007.

[21] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[22] W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Trans. Software Eng.*, 11(3):267–275, 1985.

[23] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.

[24] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, 2004.

[25] R. Könighofer and R. P. Bloem. Automated error localization and correction for imperative programs. In *FMCAD*, 2011.

[26] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis.

[27] W. R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3:1–16, 1987.

[28] W. Sack, E. Soloway, and P. Weingrad. From PROUST to CHIRON: Its design as iterative engineering: Intermediate results are important! In *In J.H. Larkin and R.W. Chabay (Eds.), Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches.*, pages 239–274, 1992.

[29] W. Sande. *Hello World! Computer Programming for Kids and Other Beginners*. Manning Publications, 2009.

[30] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

[31] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[32] E. Soloway, B. P. Woolf, E. Rubin, and P. Barth. Meno-II: An Intelligent Tutoring System for Novice Programmers. In *IJCAI*, 1981.

[33] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. *POPL*, 2010.

[34] S. S. Staber, B. Jobstmann, and R. P. Bloem. Finding and fixing faults. In *Correct Hardware Design and Verification Methods*, Lecture notes in computer science, pages 35 – 49, 2005.

[35] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, New York, NY, USA, 2010. ACM.

[36] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[37] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:183–200, 2002.