AutoRubric: Autograding Template-Based Exam Programs

Jonathon Cai

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-34
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-34.html

May 1, 2020

AutoRubric: Autograding Template-Based Exam Programs

by Jonathon Cai

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John DeNero
Research Advisor

(Date)

* * * * * * *

Professor Joshua Hug
Second Reader

(Date)

AutoRubric: Autograding Template-Based Exam Programs

Jonathon Cai
UC Berkeley
jonathon@cs.berkeley.edu

ABSTRACT

We present a system, AutoRubric, for autograding template-based student programs submitted on exams. AutoRubric automates a particularly time-consuming aspect of the grading process: assigning partial credit via a rubric specification. AutoRubric takes as input a rubric and an exam question template, as well as a set of programs to grade, and scores the programs. To address program variability, AutoRubric checks for equivalences with respect to rubric items by using SMT solvers. We present results on 1500 real student exam programs, demonstrating that in less than two minutes AutoRubric automates 43.1% of the grading effort, achieving 95.8% recall and 100% precision.

1 INTRODUCTION

Due to tremendous enrollment growth in computer science classes, computer science departments face the challenge of teaching programming at scale. In this paper, we focus on automating an important teaching task: grading student programs on exams.

Computer science exams frequently contain coding questions that evaluate student understanding of programming. In this work, we focus on grading coding questions that fit a pre-defined template. An example of such a template, adapted from a past midterm problem from UC Berkeley's introductory CS61A class, is given in Figure 1a;[1] we use this as a running example throughout our paper. The template asks the student to write Python code in five blanks, such that the completed program returns an integer that concatenates a sequence of terms produced from the function **term**: **term(1)**, . . ., **term(n)**. Figure 1c shows one possible student submission for the template in Figure 1a. The exam writer prepares a reference solution, as in Figure 1b, that represents one possible correct solution (there could be multiple reference solutions). Usually, several test cases are presented to the student in order to demonstrate what the program should emit; in Figure 1, several test cases are given in the form of Python doctests. These test cases are not necessarily exhaustive.

After the programs from an exam are collected, the course staff devises a grading scheme. One possible scheme is to grade each program against a test suite: a set of input-output pairs. Then the score would be computed according to how many tests pass. This scheme could be easily implemented by transcribing each student submission and verifying that the proper results are emitted upon execution. However, in many cases, this scheme is too strict: consider the student submission in Figure 1c. Although the student submission only differs on the first blank (Line 11) from a reference solution in Figure 1b, since the student submission causes the last term in the sequence to never be concatenated, the submission fails to pass any test cases. The student, however, has demonstrated insights that merit partial credit. We desire a grading scheme that accounts for this partial correctness.

An alternative grading scheme is to use a rubric. We can evaluate partial correctness based on what the student wrote for each blank in the template. Each blank is scored individually according to whether or not it satisfies a certain condition (typically, equivalence with respect to some other program fragment). In this way, a more fine-grained set of criteria can be applied to score student programs. Compared to the test suite, however, a disadvantage of the rubric is that it is more difficult to implement automatic scoring of a student submission; more work needs to be done than simply transcribing and then executing the program to check if it produces the right result for a given test case.

After a rubric is developed, graders manually assign partial credit to each student submission according to each item in the rubric. For an introductory class with around 1800 students, we found that grading an exam typically takes two full days, distributed across course staff. Furthermore, approximately 500 regrade requests were filed on average for each exam, indicating that manual assessments can be unreliable. This manual process is time-consuming and error-prone.

We desire a rubric grading system that satisfies several requirements:

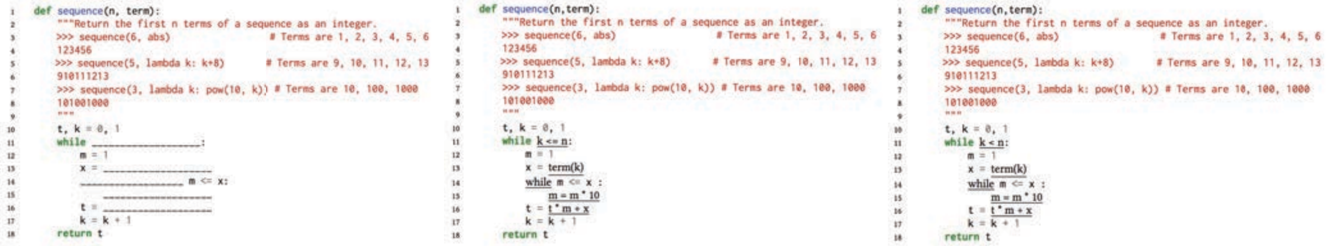
- The system must implement a rubric abstraction that graders find convenient to use. This includes support for iteratively changing the rubric as submissions are reviewed, to account for new criteria or alternative solutions.
- For each rubric item, if the system is able to automatically determine if the rubric item is satisfied, the system should score the rubric item with 100% precision. That is to say, the system should not generate any false positives. In this grading context, a false positive means that the system awards credit for a rubric item that should not actually receive credit. Any rubric item that cannot be definitively handled by the system should be reviewed by a human. This criterion is necessary in order to prevent misgrading rubric items.
- The system should score the submissions quickly.

In this paper, we design and implement a system, AutoRubric, that satisfies these requirements. To the best of our knowledge, we are the first (in this paper) to design and implement an autograder that interoperates with a rubric abstraction, for the purpose of evaluating template-based student programs.

2 RELATED WORK

In order to contextualize our contributions, we discuss, to our knowledge, prior work that is most closely related to our system. Existing techniques for evaluating student programs fall into three main categories: 1. manual grading without any computer assistance, 2. grading via test cases, and 3. semi-automated or fully

[1] https://cs61a.org/assets/pdfs/61a-fa18-mt1.pdf#page=5



```
1   def sequence(n, term):
2       """Return the first n terms of a sequence as an integer.
3       >>> sequence(6, abs)                    # Terms are 1, 2, 3, 4, 5, 6
4       123456
5       >>> sequence(5, lambda k: k+8)          # Terms are 9, 10, 11, 12, 13
6       910111213
7       >>> sequence(3, lambda k: pow(10, k)) # Terms are 10, 100, 1000
8       101001000
9       """
10      t, k = 0, 1
11      while ________________:
12          m = 1
13          x = ________________
14          ________________ m <= x:
15              ________________
16          t = ________________
17          k = k + 1
18      return t
```

(a) Template

```
1   def sequence(n,term):
2       """Return the first n terms of a sequence as an integer.
3       >>> sequence(6, abs)                    # Terms are 1, 2, 3, 4, 5, 6
4       123456
5       >>> sequence(5, lambda k: k+8)          # Terms are 9, 10, 11, 12, 13
6       910111213
7       >>> sequence(3, lambda k: pow(10, k)) # Terms are 10, 100, 1000
8       101001000
9       """
10      t, k = 0, 1
11      while k <= n:
12          m = 1
13          x = term(k)
14          while m <= x :
15              m = m * 10
16          t = t * m + x
17          k = k + 1
18      return t
```

(b) A reference solution

```
1   def sequence(n,term):
2       """Return the first n terms of a sequence as an integer.
3       >>> sequence(6, abs)                    # Terms are 1, 2, 3, 4, 5, 6
4       123456
5       >>> sequence(5, lambda k: k+8)          # Terms are 9, 10, 11, 12, 13
6       910111213
7       >>> sequence(3, lambda k: pow(10, k)) # Terms are 10, 100, 1000
8       101001000
9       """
10      t, k = 0, 1
11      while k < n:
12          m = 1
13          x = term(k)
14          while m <= x :
15              m = m * 10
16          t = t * m + x
17          k = k + 1
18      return t
```

(c) A student submission

Figure 1: Figure 1a shows a template that a student must fill out with Python code in order to return an integer that represents the concatenation of a sequence of positive integers from the function term for term(1), . . ., term(n) (for a positive integer n). Figure 1b is a reference solution, and Figure 1c is a student submission that differs only in the first blank (Line 11). If graded against a test suite, the student submission in Figure 1c would receive no credit, since it cannot pass any test cases; the resulting integer always misses the last term.

automated systems that generate fixes for student programs, which can be indirectly leveraged to assign a program score. As described earlier, the first category, manual grading, is time-consuming and often unreliable. Also established earlier, the second category of grading via test cases is usually not sufficiently fine-grained to account for partial credit. Another drawback of test cases is that a suite of test cases cannot generally prove that the student submission is truly correct: since the suite is finite, a student submission could pass all the tests in the suite but fail to cover other cases. In our experience, these kinds of submissions commonly appear. For a survey of prior work on autograding with test cases, refer to [3].

Regarding the third category, we refer to systems that provide feedback for student programs [4, 10, 11]. These systems suggest minimal repairs for student programs, attempting to fix student submissions so that they match reference solutions. In this line of work, the component that bears the strongest relation with AutoRubric is the error model language in [10]; there, it is used to describe a set of rewrite rules that captures the potential corrections for mistakes that students might make in their solutions. AutoRubric's rubric specification language, which we describe in Subsection 4.2, instead specifies a set of deterministic program fragments which are checked against each student submission for equivalence.

All these systems generate repairs with some notion of cost, and it is possible to leverage the repair cost to score the program. Lower cost repairs, meaning fewer edits are needed to reach some reference solution, would map to higher scores. We found, however, that the precision and quality of repairs generated by these systems are generally not high enough for our purposes. Furthermore, the synthesized repairs and associated costs do not align well with a consistent rubric abstraction that is convenient for a grader to use and modify. These systems were not designed for program assessment, but rather program repair, and hence do not exactly fit our needs.

We remark that the exam setting we consider in this paper differs from settings in most previous work, which focus on programs that can be freely compiled and executed, often for submission for projects and assignments for massive open online courses. In a project or assignment setting, students are free to compile and test their programs as many times as possible before submission. In this setting, students can test their programs extensively and inspect the program output. In contrast, for coding questions on computer science exams, students do not have the ability to compile or execute their programs, since most exams are handwritten.

Finally, we mention a line of complementary related work [1, 5, 6] that is relevant to our system: automatic syntax correction. In our system, we require processing raw handwritten programs into syntactically correct code, in order to invoke AutoRubric. In this paper, we perform this syntax correction process manually, but we could benefit from work on automatic syntax correction tools to alleviate this labor.

3 OVERVIEW

We now present an overview of the workflow of the AutoRubric system. The major phases are outlined in Figure 2: (1) Program Transcription, (2) Program Translation, (3) Rubric Synthesis, and (4) Equivalence Checking. At a high level, our strategy for assigning partial credit to each student submission is to fragment the program into individual components, each of which corresponds to what the student wrote in each blank of the template, and then to use these components to verify that rubric items are satisfied. In order to do this, we require (1) a representation of the student submission suitable for verification, (2) a representation of each rubric item suitable for verification, and (3) a back-end engine that performs the verification. These three requirements roughly correspond to the three phases of program translation, rubric synthesis, and equivalence checking.

(1) **Program Transcription.** The program transcription phase converts raw handwritten programs, which may contain syntax errors, into syntactically correct programs. Syntactic correctness is a common requirement for program analysis tools; this condition is needed in order to access the abstract syntax tree for the program. The program transcription phase rejects programs that either do not match the original template or which are too difficult to fix syntactically. These syntactically incorrect submissions must be reviewed by a human in order to receive partial credit. In our work, we do not implement this phase automatically and instead review each submission manually to fix syntax errors.

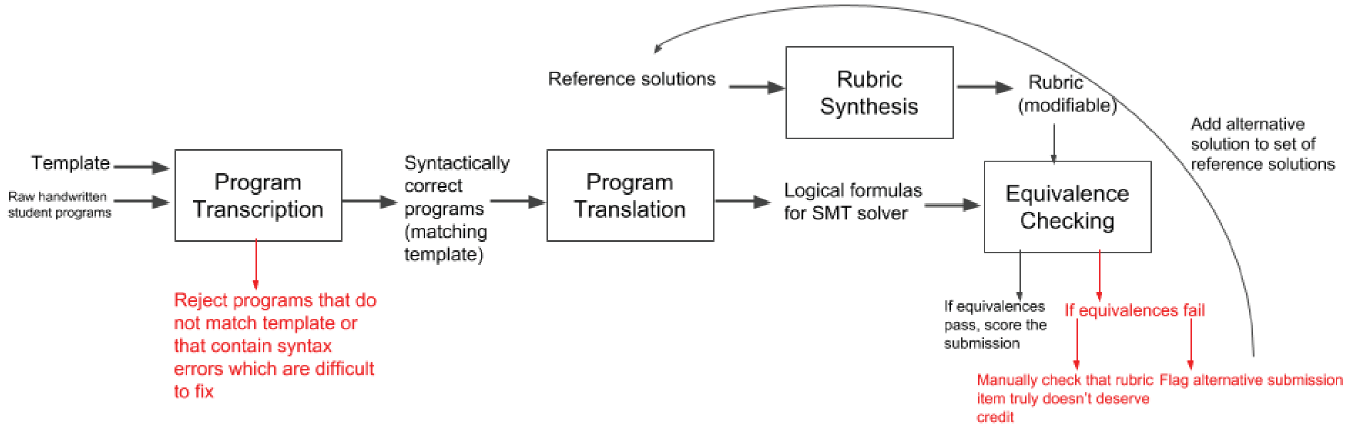

Reference solutions
Rubric Synthesis
Rubric (modifiable)
Add alternative solution to set of reference solutions
Template
Raw handwritten student programs
Program Transcription
Syntactically correct programs (matching template)
Program Translation
Logical formulas for SMT solver
Equivalence Checking
Reject programs that do not match template or that contain syntax errors which are difficult to fix
If equivalences pass, score the submission
If equivalences fail
Manually check that rubric item truly doesn't deserve credit
Flag alternative submission

Figure 2: Overall workflow of AutoRubric

In our work, this manual process was performed in two stages: firstly, we requested that students transcribe their own solutions to four exam problems, submitting them via a form, and secondly, a single human reviewer cross-checked the transcriptions against the actual student exams. Approximately 1500 students submitted their transcriptions via the form. Assuming a conservative estimate of 10 minutes for each transcription (of four programs), the transcription time would be 250 hours total for 1500 students. It took approximately 20 hours for the reviewer to cross-check 500 of these student transcriptions. The reviewer checked every single blank in each submission and applied fixes when needed to ensure the transcription matched the original exam.

(2) **Program Translation.** The program translation phase takes each student submission and fragments the program into individual components, each of which corresponds to what the student wrote in each blank of the template. Each of these components is translated into logical formulas that can be passed into a satisfiability modulo theory (SMT) solver during the equivalence checking phase. In order to implement this phase, we built a source-to-source compiler that converts Python (the submission language) into Z3 [2] (the SMT logical formula language).

(3) **Rubric Synthesis.** The rubric synthesis phase converts a set of reference solutions into a specification file that represents a rubric. Initial rubric items are synthesized according to the contents of the reference solutions that correspond to template blanks. Listing 1 shows an example. After the specification file is synthesized, the grader can modify the rubric in custom ways. For instance, the grader could create a rubric item that awards partial credit for the **k < n** answer in Figure 1c, as in Line 3 of Listing 7. We designed a simple specification language that the grader can use to specify the rubric, which supports Python fragments, boolean logic operators, and references to prior rubric items.

(4) **Equivalence Checking.** The equivalence checking phase takes each rubric item and converts it into a logical formula that a SMT solver can use to check the equivalence of each rubric item with the relevant portion of a student submission. If the rubric contains multiple reference solutions, the student submission is checked against all reference solutions, and ultimately, the student submission is assigned the largest score from all the reference solutions. A human can manually review submissions with particularly low scores that pass a significant number of test cases, identifying if the submissions might be alternative solutions. If the submission is an alternative solution, it should be added to the set of reference solutions. The rubric synthesis and equivalence checking phases can then be restarted. These phases can be run iteratively till the graders are satisfied with AutoRubric's results.

In later sections, we focus on details regarding the latter three phases (program translation, rubric synthesis, and equivalence checking), since they form the core of AutoRubric.

4 IMPLEMENTATION

Our system currently only handles Python programs, converting them to logical formulas for the Z3 SMT solver. In the future, we hope to extend our architecture to other front and back ends.

We chose to use a SMT solver because it can prove the validity of first-order formulas in a large number of built-in logical theories. In particular, we use the SMT solver to verify equalities in linear arithmetic, which commonly appear when grading.

4.1 Program Translation

To translate a student program into a representation suitable for verification with rubric items, we implemented a source-to-source compiler from Python to Z3. Z3 supports statements such as

```
prove(e1 == e2)
```

for some Z3 expressions `e1` and `e2`. If the statement `e1 == e2` is true and if Z3 can verify it, then Z3 emits `"proved"`; otherwise, it either hangs or emits a counterexample. For example, if `e1` is `x + f(k + 0)` and `e2` is `f(k) + x`, then `prove(e1 == e2)` should emit `"proved"`. Here, `x`, `k`, and `f` are terms in Z3; specifically `x` is an

Int, k is an Int, and f is an uninterpreted function mapping Int to Int. Int is a type built into Z3.

In order to extract relevant portions of Python abstract syntax trees, we used the Lark parser.[2] We support translation of a subset of the Python language, including numeric constants, variables, comparisons, boolean logic, most arithmetic expressions, function calls, assignment statements, return statements, and comma statements. Notably, we do not yet support lambda expressions. In the future, it may be possible to implement support for reasoning over more sophisticated Python data structures like sets and lists. [7]

Most Python expressions are structurally very similar to their Z3 analogs, and so most translations are straightforward. We mention some non-standard choices and simplifications we made. Python assignment statements, such as k = 2, are translated into Z3 uninterpreted function calls on the function SPECIAL_EQ such as SPECIAL_EQ(k, 2), since there is no notion of assignment in Z3. Python augmented arithmetic statements, such as m *= 10, are translated into their expanded Z3 analogs such as m == m * 10. The Python **return** statement is treated as a Z3 uninterpreted function. Finally, we make the simplification that all variables are integers and all uninterpreted functions map their (integer) arguments to integers. This leads to Z3 expressions that are not totally semantically faithful, for instance, for Python programs involving floating point, but we found this simplification mostly sufficient for our purposes.

4.2 Rubric Synthesis

In order to implement a rubric abstraction, we designed a rubric specification language embedded inside YAML, a data serialization language. In designing this rubric specification language, we prioritized usability: the rubric interface should be convenient to use and modify for a grader, not requiring knowledge of formal methods or the underlying verification processes.

We implemented a rubric synthesizer that takes as input a Python reference solution and synthesizes an initial rubric in our specification language. An example of an initial rubric, synthesized from the reference solution in Figure 1b, is given in Listing 1. Examples of more complex rubrics, which were modified from the initially synthesized ones, are given in Listings 6 and 8.

Each rubric item is specified by three attributes. The first item attribute is the Python fragment, possibly intermixed with operators from the rubric specification language. The second item attribute defines rubric item parameters; currently two parameters are supported. The first (required) parameter is the blank number, used to reference content from a particular template blank to match. The second (optional) parameter is explicit specification of a mode: a mode is used to define the way the match between the rubric code fragment and student code fragment is performed. The third item attribute is the score associated with the rubric item.

In aggregate, the first and second item attributes define the Python fragment, on a given blank, that the student submission must match in order to receive credit for the rubric item corresponding to the third item attribute, the score. The program translator described in the previous subsection is used to convert Python fragments in the first item attribute into Z3 logical formulas.

```
rubric1:
    - ['k <= n', 'Blank 1', 'Score 1.0']
    - ['term ( k )', 'Blank 2', 'Score 1.0']
    - ['while', 'Blank 3', 'Score 1.0']
    - ['m *= 10', 'Blank 4', 'Score 1.0']
    - ['t * m + x', 'Blank 5', 'Score 1.0']
```

Listing 1: Initially synthesized rubric corresponding to Figure 1b. The default assigned score is 1.0.

For the first rubric item attribute, we include support for several operators. We use the || operator to denote multiple options for code fragments that the student submission can match in order to receive credit for the rubric item. The ordering matters: the options are tested in the left-to-right order given in the first item attribute, and if a particular Python fragment is matched, the evaluation short circuits and no further options are tested. The rubric specification language also supports referencing truth values associated with prior rubric items, using the ITEM_N_BOOL syntax to refer to rubric item N (1-indexed). That is to say, for a given submission, if the first rubric item is satisfied, ITEM_1_BOOL would evaluate to true. Furthermore, these truth values can be combined using boolean logic operators (_AND and _OR), such as in Line 6 of Listing 8. These boolean logic operators are useful in the event that the grader wants to define a rubric item dependent on truth values of other rubric items.

Regarding the second parameter of the second item attribute, the mode, four modes are supported. Mode.EXACT declares that the rubric code fragment, verbatim, must exactly match the student code fragment; Mode.CONTAIN declares that the rubric code fragment, verbatim, should be contained within the student code fragment; Mode.PARTIAL declares that the rubric code fragment should be contained within the student code fragment (up to equivalence); and Mode.EQUIV (the most common mode) declares that the rubric code fragment should be equivalent to the student code fragment, which the SMT solver can hopefully verify. If the second parameter is left out, the mode is inferred: if the compiler from the previous subsection can handle the rubric code fragment, Mode.EQUIV is used by default; if not, Mode.CONTAIN is used.

4.3 Equivalence Checking

After the prior two phases are performed, we are left with Z3 statements of the form prove(r == s), where r denotes a rubric code fragment and s denotes a student code fragment. Z3 evaluates these statements, awarding credit according to each rubric item's truth value.

If the equivalence check passes for a rubric item, then a human need not review it, because AutoRubric has produced a proof of equivalence between the rubric and student code fragments. If the equivalence check fails for a rubric item, the item must be manually reviewed by a human to check that it is indeed false. This is because AutoRubric does not provide a guarantee that the submission code fragment is truly incorrect; it is possible that AutoRubric failed to perform the verification due to limitations of the implementation, or the rubric item was under-specified. At this point, when reviewing submissions manually, alternative solutions can be flagged. An

[2] https://github.com/lark-parser/lark

```
1   def sequence ( n , term ) :
2       t , k = 0 , 1
3       while k <= n :
4           m = 1
5           x = term ( k ) // 10
6           while m <= x :
7               m *= 10
8           t = t * m * 10 + term ( k )
9           k = k + 1
10      return t
```

Listing 2: Alternative solution for problem presented in Figure 1

```
rubric1:
    - ['k <= n', 'Blank 1', 'Score 1.0']
    - ['term ( k ) // 10', 'Blank 2', 'Score 1.0']
    - ['while', 'Blank 3', 'Score 1.0']
    - ['m *= 10', 'Blank 4', 'Score 1.0']
    - ['t * m * 10 + term ( k )', 'Blank 5', 'Score 1.0']
rubric2:
    - ['k <= n', 'Blank 1', 'Score 1.0']
    - ['term ( k )', 'Blank 2', 'Score 1.0']
    - ['while', 'Blank 3', 'Score 1.0']
    - ['m *= 10', 'Blank 4', 'Score 1.0']
    - ['t * m + x', 'Blank 5', 'Score 1.0']
```

Listing 3: Updated rubric for problem in Figure 1

example of a flagged alternative solution, for the problem in Figure 1, is shown in Listing 2.

In this alternative solution, the computation of m, representing the number of digits needed to store the next concatenated term, differs from the reference solution in Figure 1b in Lines 5 and 8. Specifically, in the alternative solution, x is a factor of 10 below x in Figure 1b, but the factor of 10 is ultimately included in Line 8. After we convince ourselves this is an alternative solution, we can add it to the set of reference solutions and re-synthesize a rubric, presented in Listing 3, which can then be modified as desired. In this way, we may run AutoRubric iteratively till reaching a satisfactory stopping point.

Figure 3: rect

		Predicted	
		True	False
Actual	True	1941	69
	False	0	1702

Figure 4: sequence

		Predicted	
		True	False
Actual	True	1661	90
	False	0	889

Figure 5: repeat_digits

		Predicted	
		True	False
Actual	True	1134	51
	False	0	1551

Figure 6: aggregate

		Predicted	
		True	False
Actual	True	4736	210
	False	0	4142

Figure 7: Confusion matrices for rubric items for syntactically correct, unproblematic programs

5 RESULTS

5.1 Performance on Real Student Programs

We evaluate AutoRubric on actual student programs from Midterm 1 of CS61A,[3] an introductory computer science class at UC Berkeley, from the Fall 2018 semester. From approximately 1800 students, we selected a random subset of 500 students. For each of these students, we used AutoRubric to autograde three midterm coding problems, named **rect**, **sequence**, and **repeat_digits**,[4] amounting to 1500 programs in total. **sequence** was presented in Figure 1. The templates for **rect** and **repeat_digits** are shown in Listings 4 and 5. This set of 1500 programs contains a number of syntactically incorrect programs. 30, 60, and 43 programs were syntactically incorrect for **rect**, **sequence**, and **repeat_digits**, respectively. Furthermore, 6 and 1 programs caused our tool to crash or hang for **rect** and **repeat_digits**, respectively. We believe these problematic behaviors were caused by limitations of the Z3 SMT solver, which occasionally is neither able to verify an equivalence nor produce a counterexample for certain inputs. Our tool cannot handle these programs, and so all these programs would need to be inspected manually. We include these problematic programs in our final performance results in order to accurately account for the amount of labor that our tool would save.

In order to assess our tool, we required ground truth: we assumed that the actual scores assigned by graders (after resolving all regrade requests) were perfect. The actual rubrics for each of the three problems, formatted on Gradescope [9], are presented in Figures 8, 9, and 10. We transcribed each rubric as faithfully as possible, using our rubric specification language. The transcribed rubrics are shown in Listings 6, 7, and 8, containing 8, 6, and 6 rubric items, respectively. We than ran AutoRubric on the set of selected student programs.

We report AutoRubric's performance on syntactically correct, unproblematic programs, presenting a confusion matrix for the rubric items for each problem in Figures 3, 4, and 5; we also present a confusion matrix for the aggregate results in Figure 6. In this grading setting, a true positive indicates that AutoRubric marked a rubric item true that a grader also marked true.

The aggregate recall was 95.8%, and the aggregate precision was 100%. Note that the precision was 100%, since due to the nature of the SMT solver, AutoRubric cannot mark a rubric item true unless it can provide a proof of equivalence. Items that AutoRubric marks true do not need to be reviewed by humans. Since there are 500*(8+8+6), or 11000, rubric items in total, and AutoRubric marks 4736 of them true, AutoRubric save about 43.1% of the total labor. In practice, human graders would have to check the other 56.9% of rubric items to determine if they are indeed false. This estimate is conservative, since it does not take into account that certain rubric item truth values are mutually exclusive (such as rubric items 1 and 2 in Listing 7.)

The total execution time for all 1500 programs was less than two minutes.

AutoRubric produced false negatives due to limitations of the current implementation. For example, for **rect**, some submissions

[3] https://cs61a.org/

[4] https://cs61a.org/assets/pdfs/61a-fa18-mt1.pdf

included expressions involving floating point numbers, and AutoRubric currently lack support for floating point reasoning. Additionally, AutoRubric cannot handle uninterpreted functions and unusual expressions that are not included in the rubric. Iterating on the rubric as alternative solutions are flagged, as presented earlier, would improve performance. Also mentioned earlier, AutoRubric lacks support for certain Python language features like lambda expressions.

5.2 Rubric Transcription Discrepancies

Note that the rubrics were not transcribed perfectly.

For **rect**, we adapted expressions from the original rubric items 4 and 5. Specifically, we used the expressions
`other == round(area / side)` and
`other == round (perimeter / 2 - side)` in place of
`area / side == round(area / side)` and
`perimeter /2 - side == round (perimeter / 2 - side)`, respectively. We made these substitutions, because presently AutoRubric is unable to reason about transitive relationships between variables. Furthermore, for rubric item 5, we included the expression `2*(side+other) == perimeter`, since this was in the original exam reference solution. We also omitted a rubric item "Fully correct solution even though `other` is used in an unexpected way", not shown in Figure 8, that was applied very rarely (to 4 out of approximately 1800 submissions), because it did not have a clear definition we could transcribe.

For **sequence**, there was a very uncommonly used rubric item "[Minor error] off by one digit", not shown in Figure 9, which was applied to 41 out of approximately 1800 exam submissions. This item could not be localized to any particular set of blanks, so we did not transcribe it.

For **repeat_digits**, we did not transcribe the expressions involving lambda expressions from the first rubric item, since AutoRubric does not yet support lambda expressions.

Finally, we did not implement the "forbid" feature, which forbids certain expressions from appearing, as in rubric item 4 for Figure 8 and rubric item 6 for Figure 9.

6 LIMITATIONS AND FUTURE WORK

We were motivated to build AutoRubric in order to improve the exam grading process for an introductory class we teach: CS61A at UC Berkeley. The main pragmatic barrier that prevents us from deploying our system for a real exam grading setting is that the program transcription phase of our system is manually intensive and time-consuming. In future work, in order to alleviate this labor, we hope to invest in developing an optical character recognition (OCR) system to automate the transcription process from scanned handwritten programs into actual programs, possibly incorporating a syntax correction module from prior work such as [1, 5, 6]. An alternative, avoiding the OCR system, is to have students take exams electronically; one such system is BlueBook [8].

We hope to further extend our source-to-source compiler from Python to Z3, as well as our rubric specification language. It should be possible to easily implement features like transitive reasoning and support for other Python expressions like lambda expressions.

We are interested in running user studies in order to observe how graders use AutoRubric in a real-time grading setting. In particular, we are interested in how graders would iteratively update the rubric.

In this work, we focused on template-based coding questions, mainly because they are the format for coding questions for CS61A at UC Berkeley. In future work, we are interested in considering how to extend our system to grade programs with more variable structure.

7 DISCUSSION

In this paper, we presented the AutoRubric system for autograding template-based exam programs. It employs the Z3 SMT solver to verify equivalences regarding program fragments of interest. We built a source-to-source compiler from Python to Z3 in order to facilitate the Z3 theorem proving process. We also designed a rubric specification language that makes it convenient to specify and modify rubrics. We have evaluated AutoRubric on 1500 real student programs, demonstrating that it can automate 43.1% of the grading effort and achieve 95.8% recall and 100% precision, while executing in less than two minutes. Our results show that AutoRubric is effective and fast, and we believe that AutoRubric can provide a basis for autograding template-based programs to thousands of students.

ACKNOWLEDGMENTS

We would like to thank Gregory Jerian for helping with cleaning the dataset.

REFERENCES

[1] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

[2] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[3] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic Test-based Assessment of Programming: A Review. *J. Educ. Resour. Comput.* 5, 3, Article 4 (Sept. 2005). https://doi.org/10.1145/1163405.1163409

[4] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 465–480. https://doi.org/10.1145/3192366.3192387

[5] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep Reinforcement Learning for Programming Language Correction. arXiv:arXiv:1801.10467

[6] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[7] Edmund S. L. Lam and Iliano Cervesato. 2014. Reasoning about Set Comprehension. In *In SMT'14*.

[8] Chris Piech and Chris Gregg. 2018. BlueBook: A Computerized Replacement for Paper Tests in Computer Science. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 562–567. https://doi.org/10.1145/3159450.3159587

[9] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17)*. ACM, New York, NY, USA, 81–88. https://doi.org/10.1145/3051457.3051466

[10] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *SIGPLAN Not.*

48, 6 (June 2013), 15–26. https://doi.org/10.1145/2499370.2462195

[11] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 481–495. https://doi.org/10.1145/3192366.3192384

```python
def rect(area, perimeter):
    """Return the longest side of a rectangle with area and perimeter that has integer sides.

    >>> rect(10, 14)  # A 2 x 5 rectangle
    5
    >>> rect(5, 12)   # A 1 x 5 rectangle
    5
    >>> rect(25, 20)  # A 5 x 5 rectangle
    5
    >>> rect(25, 25)  # A 2.5 x 10 rectangle doesn't count because sides are not integers
    False
    >>> rect(25, 29)  # A 2 x 12.5 rectangle doesn't count because sides are not integers
    False
    >>> rect(100, 50) # A 5 x 20 rectangle
    20
    """
    side = 1
    while side * side _____________ area:
        other = round(_____________)
        if _____________:
            _____________
        side = side + 1
    return False
```

Listing 4: Template for rect

```python
def repeat_digits(n):
    """Print the repeated digits of non-negative integer n.

    >>> repeat_digits(581002821)
    2
    0
    1
    8
    """

    f = _____________

    while n:

        f, n = _____________, _____________
```

Listing 5: Template for repeat_digits

```
rubric1:
    - ['<=', 'Blank 1, Mode.EXACT', 'Score 1.0']
    - ['<', 'Blank 1, Mode.EXACT', 'Score 0.5']
    - ['( perimeter / 2 ) - side || area / side', 'Blank 2', 'Score 0.5']
    - ['side * other == area || other == round(area / side) || area % side == 0', 'Blank 3, Mode.PARTIAL', 'Score 1.0']
    - ['side + other == perimeter / 2 || other == round ( perimeter / 2 - side) || 2*(side+other) == perimeter', 'Blank 3, Mode.PARTIAL', 'Score 1.0']
    - ['and', 'Blank 3, Mode.CONTAIN', 'Score 0.5']
    - ['return other || return max(side, other) || return max(other, side)', 'Blank 4', 'Score 1.0']
    - ['return side', 'Blank 4', 'Score 0.5']
```

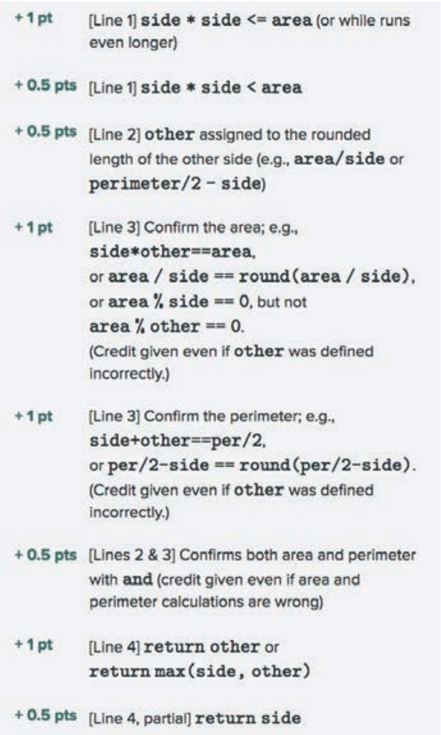
Listing 6: Transcribed rubric for rect

```
rubric1:
    - ['k <= n', 'Blank 1', 'Score 1.0']
    - ['k < n', 'Blank 1', 'Score 0.5']
    - ['term ( k )', 'Blank 2', 'Score 1.0']
    - ['while', 'Blank 3', 'Score 1.0']
    - ['m *= 10', 'Blank 4', 'Score 1.0']
    - ['t * m + x || t*m +term(k)', 'Blank 5', 'Score 1.0']
```

Listing 7: Transcribed rubric for sequence

```
rubric1:
    - ['repeat || repeat(-1)', 'Blank 1', 'Score 1.0']
    - ['repeat( n % 10 )', 'Blank 1', 'Score 1.0']
    - ['f( n % 10 )', 'Blank 2', 'Score 0.5']
    - ['f((n // 10) % 10)', 'Blank 2', 'Score 0.5']
    - ['( ITEM_1_BOOL _AND ITEM_3_BOOL ) _OR ( ITEM_2_BOOL _AND ITEM_4_BOOL )', 'Multiple', 'Score 0.5']
    - ['n // 10', 'Blank 3', 'Score 1.0']
```

Listing 8: Transcribed rubric for repeat_digits



+ 1 pt [Line 1] **side * side <= area** (or while runs even longer)

+ 0.5 pts [Line 1] **side * side < area**

+ 0.5 pts [Line 2] **other** assigned to the rounded length of the other side (e.g., **area/side** or **perimeter/2 - side**)

+ 1 pt [Line 3] Confirm the area; e.g., **side*other==area**, or **area / side == round(area / side)**, or **area % side == 0**, but not **area % other == 0**. (Credit given even if **other** was defined incorrectly.)

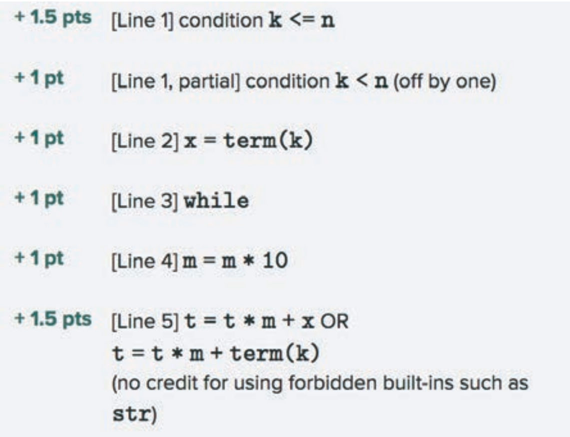
+ 1 pt [Line 3] Confirm the perimeter; e.g., **side+other==per/2**, or **per/2-side == round(per/2-side)**. (Credit given even if **other** was defined incorrectly.)

+ 0.5 pts [Lines 2 & 3] Confirms both area and perimeter with **and** (credit given even if area and perimeter calculations are wrong)

+ 1 pt [Line 4] **return other** or **return max(side, other)**

+ 0.5 pts [Line 4, partial] **return side**

Figure 8: Actual Gradescope rubric for rect



+ 1.5 pts [Line 1] condition **k <= n**

+ 1 pt [Line 1, partial] condition **k < n** (off by one)

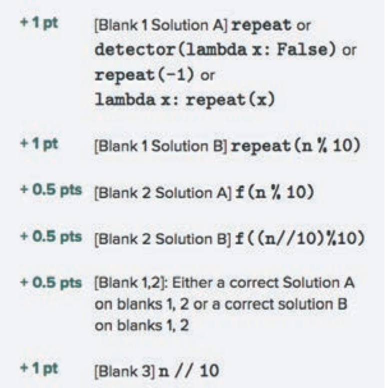
+ 1 pt [Line 2] **x = term(k)**

+ 1 pt [Line 3] **while**

+ 1 pt [Line 4] **m = m * 10**

+ 1.5 pts [Line 5] **t = t * m + x** OR **t = t * m + term(k)** (no credit for using forbidden built-ins such as **str**)

Figure 9: Actual Gradescope rubric for sequence



+ 1 pt	[Blank 1 Solution A] `repeat` or `detector(lambda x: False)` or `repeat(-1)` or `lambda x: repeat(x)`
+ 1 pt	[Blank 1 Solution B] `repeat(n % 10)`
+ 0.5 pts	[Blank 2 Solution A] `f(n % 10)`
+ 0.5 pts	[Blank 2 Solution B] `f((n//10)%10)`
+ 0.5 pts	[Blank 1,2]: Either a correct Solution A on blanks 1, 2 or a correct solution B on blanks 1, 2
+ 1 pt	[Blank 3] `n // 10`

Figure 10: Actual Gradescope rubric for repeat_digits