Automatically Generating Programming Questions Corresponding to Rubrics Using Assertions and Invariants

Masami Hagiya, Kosuke Fukuda, Yoshinori Tanabe, Toshinori Saito

Automatically Generating Programming Questions Corresponding to Rubrics Using Assertions and Invariants

Masami Hagiya[1] Kosuke Fukuda[1] Yoshinori Tanabe[2] Toshinori Saito[3]

[1] Graduate School of Information Science and Technology, University of Tokyo
[2] School of Literature, Tsurumi University
[3] Graduate School of Practitioners in Education, Seisa University

tanabe-y@tsurumi-u.ac.jp

Abstract. The importance of programming in the context of primary and secondary education is increasing, reflecting incorporation of programming in the reformation of Japanese curriculum guidelines for schools, the release of which is planned in 2020. Earlier research developed proposed rubrics for assessment of student attainment levels in various fields of high-school informatics, including programming. The reformation requires that university entrance examinations should include informatics; it has thus become necessary to create a large number of questions exploring various levels of attainment. Here, we develop methods automatically generating questions in the field of programming; we use program correctness proofs and a theorem prover to this end. Questions corresponding to each level are automatically generated from a single program featuring a correctness proof by analyzing the source code and the proof with the aid of a theorem prover. These methods can help prepare questions for university entrance examinations, and can also be applied to E-learning systems when questions are used to allow evaluation of student attainment levels. We conducted a small experiment, and the results show that this is a promising approach.

Keywords: Programming education, Rubric, Examination, Correctness proof, Invariant, Assertion

1 Introduction

The teaching of computer science and information technology, or informatics in general, is increasingly common not only in higher educational institutions but also in primary and secondary schools. This world-wide trend is especially prominent in Japan. The Japanese government has decided to introduce computer programming in primary schools in the next standard curriculum guideline that will take effect in 2020 [1]. The subject "Informatics I" of the high school curriculum will also be modernized to include programming as a mandatory field of study (this is presently optional) [2, 3]. Given this trend, it is becoming increasingly important to assess student understanding of, and skill in, informatics. In a research project funded by the government and led by Osaka University [4], attainment levels for each field of high school informatics, including programming, have been formulated as "rubrics", and methods for assessment of student performance using these rubrics are under development.

To assess student attainment levels, it is necessary to create appropriate questions for each level of all rubrics. Obviously, such questions must be posed during high school education to allow adjustment of teaching methods and materials with reference to student attainment levels. Such questions can also be used when evaluating admission to Japanese universities. The government is now investigating the possibility of introducing Informatics to the common entrance examination run by the Center for University Entrance Examinations.

1.1 Goal

Here, we develop methods that automatically generate questions in the field of programming to assess students' understanding of programs; we use correctness proofs and a theorem prover to this end. Questions corresponding to each level of rubrics mentioned above are automatically generated from a single program featuring a correctness proof by analyzing the source code and the proof with the aid of a theorem prover. Our assumed target is the common entrance examination of Japanese universities mentioned in the previous section. Therefore, it should be possible to generate a large number of questions. Also, because the score of the examination need to be automatically evaluated, the format of questions should be true or false, multiple choice, or the like.

To achieve our goal, we use the methods of software engineering, including software verification and testing, to automatically generate programming questions; this is a field of study within Informatics. Although these technologies were developed to verify or test programs written by humans, we use them to test humans. These two applications are not contradictory. Advanced software verification and testing algorithms are intended to replace humans who test and debug programs. We therefore apply these technologies to test whether humans (i.e., students) have the required knowledge and skill for testing and debugging programs.

Here, we assume that a program features a correctness proof. Questions are then automatically generated corresponding to each level of rubrics. It is therefore not necessary to create different questions for different levels of the rubrics.

A correctness proof is obtained by reference to the given program pre-condition and post-condition; appropriate invariants of while-statements and lemmas are also given and referenced to derive the post-condition from the pre-condition. Our hypothesis is that invariants used in the correctness proof of a program play an important role in understanding the program. We do not require even those who make questions of examinations to write correctness proofs. It is sufficient for them to provide appropriate loop invariants.

For question generation, we also use the theorem prover as a Satisfiability Modulo Theories (SMT) solver. When appropriate constraints are automatically generated from the program and the correctness proof, the SMT solver solves the constraints and produces inputs or variable assignments that are used to generate questions.

This method allows the preparation of questions not only for university admission examinations but also for E-learning systems where the questions must reflect differences in student abilities.

1.2 Rubrics

Matsunaga and Hagiya have formulated rubrics for the attainment levels of each field of high-school informatics [5] as part of the research project (mentioned above) led by Osaka University [4]. The suggested rubrics for programming are shown in Table 1.

At the lowest level 1-1, students can recognize the description of a given program, and can thus understand the structure of a program (such as declarations and if-statements) by reference to the grammar of the programming language. At the next level 1-2, at which the first major level is entered, students can trace the program after a given input. At this level, students are required to understand the meaning of each construct of a program and mechanically execute the program given a concrete input. At level 2-1, students can explain what a given program does. This requires that they understand the goal or specification of the program and how the program meets that specification. At level 2-2, in addition to correctly understanding programs, students can modify a given program to attain a different goal or specification, or repair an incorrect program to meet a given specification.

Table 1. Rubrics for Programming

1-1	being able to recognize the description of a given program
1-2	being able to trace the execution of a given program
2-1	being able to explain what a given program does
2-2	being able to modify or debug a given program to solve a given problem
3	being able to design an algorithm and write a program to solve a given problem
4	being able to improve an algorithm or a program to solve a given problem more effectively
5	being able to evaluate and improve a program during program construction

1.3 Approach

By reference to the rubrics explained above, we took the following approach to automatically generate questions corresponding to the three levels of the rubrics.

As Rubric 1-2 requires students to trace the execution of a given program, we use questions that ask students to explain what will happen when the program is executed after a given input. We have developed a method that generates an input, after which the program follows a given (appropriate) execution path.

Rubric 2-1 requires students to explain what a given program does. Such an explanation of functionality requires comprehension of the assertion that is always true at each step of the program. Therefore, we use questions that ask students to distinguish between states that satisfy the assertion and those that do not. We have developed a method that generates two kinds of states at any given point of the program.

For Rubric 2-1, we also use questions to see if students understand dynamic behavior of a program. Those questions give a pair of states at a point of the program and ask students if one state of the pair can precede the other in a certain execution path.

Third kind of question for Rubric 2-1 gives a list of conditions and asks students if each condition in the list is always true during execution of the program.

As Rubric 2-2 requires students to repair an incorrect program, we use questions that ask students to fill a blank in the code by choosing the correct expression from certain candidate expressions. Our method generates appropriate incorrect expressions.

We mainly focus on the above three levels of the rubrics because these levels are considered relatively important in high school education and for common entrance examination.

2 Related Work

A great deal of effort has been devoted to the development of methods that automatically generate questions for use in programming education; it is very difficult to exhaustively review the systems that implement such methods. However, most systems, including those of Radošević, et al. [6] and Wakayama and Maeda [7], are based on question templates. Therefore, different templates must be prepared for different kinds of questions.

In contrast to such systems, we do not use question templates but, rather, programs with correctness proofs. Furthermore, a single program with a correctness proof can be used to generate different kinds of questions; thus, those for the different rubric levels in the present instance.

As related work, we could mention assessment systems which automatically test and grade submissions by students [8]. Since they evaluate programs written for a given goal, they mainly evaluate student ability of designing algorithms and writing programs, and correspond to Rubric 3 and higher levels.

3 Methods

This section presents the methods that we use and example questions generated by these methods.

We detail the methods for Rubric 2-1 and outline the others, because examining students understanding of a program is the most challenging among the three levels of the rubrics.

Although our methods are applicable to any programming language, we adopt a subset of Python as our target language because (1) Python is currently widely recognized as a common language for teaching programming to novices, (2) we actually teach Python in some classes of our schools, and (3) various tools for formal verification and symbolic execution of Python programs are available. In particular, we employ PyExZ3 [9], a tool for dynamic symbolic execution of Python programs, for generating questions for Rubric 2-1.

As mentioned in the introduction, we assume that a program features a correctness proof, which is obtained from the given program pre-condition and post-condition together with appropriate assertions embedded in the program. Invariants are typical assertions that are put in loop statements.

Although a theorem prover is required to obtain a correctness proof, generation of questions for the three levels of the rubrics in this paper is possible only with appropriate assertions embedded in the program, as far as those assertions are correct. Therefore, the methods can be used with limited knowledge about formal verification or theorem proving.

3.1 Rubric 1-2

Rubric 1-2 indicates the level at which students can correctly trace the execution of a given program. To generate questions, our tool finds an input to the program that satisfies the program pre-condition and produces a path when the program is executed with this input. An execution path is presented to the tool as a sequence of truth values indicating the conditions imposed on if-statements and while-statements along the path.

To find such an input, the tool generates the path condition under which execution of the code should follow the given path. The tool does this by examining the if-statements and while-statements along the path. This process is similar to the symbolic execution of Pex [10]. Although Pex automatically searches for paths to take, the tool views the path instructions as inputs and follows them, which allows question difficulty to be controlled.

The tool then finds a solution featuring conjunction of the generated path condition and the program pre-condition, using the SMT solver Z3 [11, 12].

3.2 Rubric 2-1

Rubric 2-1 indicates the level at which students can explain the behavior of a given program. We assume that the post-condition can be proven from the pre-condition with the help of invariants and lemmas. Consequently, each statement of the program satisfies a certain assertion, which is derived from the program post-condition and the loop invariants of while-statements in the program. If the program commences with an input satisfying the pre-condition, the assertion of each statement holds prior to execution. We thus reduce the ability to explain the behavior of a program to that of comprehending the assertion of each statement in the program.

To determine whether students comprehend an assertion in the program, we take the following approach. Those inputs that satisfy the program pre-condition always satisfy the program assertions whereas states that violate a program assertion can never develop from a correct input. If students can discriminate correct from incorrect states, they are considered to comprehend the assertions and hence exhibit some understanding as to why the program behaves correctly.

To generate inputs that satisfy the pre-condition, the tool automatically creates an execution path that leads to the specified point of the program. The tool then traces the path and generates the path condition via symbolic execution, obtaining an input.

Incorrect states are those falsifying conditions confirmed to be valid. Since all correct states satisfy the conditions, we generate incorrect states by modifying correct ones. We make as few modifications as possible, so that obtained states appear to be correct. The SMT solver is called to find the new value that falsifies the conditions.

As for dynamic behavior, we choose conditions that are always satisfied by a pair of states along an execution path of a program. Correct pairs are obtained from actual execution paths of the program. Incorrect pairs are obtained by modifying correct ones so that the conditions are falsified.

Finally, incorrect conditions are obtained by modifying correct ones. For a candidate condition obtained from a correct one, we search for an execution path of a program that actually falsifies the condition.

The questions generated by the method for the bubble sort program are shown below.

Question 1: Assume that the function in Figure 1 is executed and reaches the point L1. Answer whether the designated combination of values is possible or not.

(1) `len(lst)=6, i=2, j=5, lst[1]=8, lst[3]=2.`

(2) `len(lst)=6, i=1, j=4, lst[2]=5, lst[5]=5.`

The answers are: (1) is IMPOSSIBLE and (2) is POSSIBLE.

```
def bubblesort(lst):
  for j in range(len(lst) - 1):
    for i in range(len(lst) - 1 - j):
      if lst[i] > lst[i + 1]:
        lst[i + 1], lst[i] = lst[i], lst[i + 1]
      # L1
    # L2
```

Fig. 1. Source for bubble sort function

Question 2: Answer whether both of the following two conditions can be satisfied or not.

The function in Figure 1 reaches the point L1 and has the combination of values shown at [Before].

The function continues to run and reaches the point L1 again to have the combination of values shown at [After].

(1) [Before] len(lst)=6, j=3, lst[4]=4; [After] j=2, lst[4]=8

(2) [Before] len(lst)=6, j=5, lst[2]=4; [After] j=1, lst[2]=2

The answers are: (1) is IMPOSSIBLE and (2) is POSSIBLE.

Question 3: Assume the function reaches the point L2 and 2 <= j <= len[lst] - 1 holds. Answer whether each of the following is always correct or not.

(1) lst[j-2] <= lst[j-1]

(2) lst[j-1] <= lst[j]

The answers are: (1) is NOT ALWAYS CORRECT and (2) is ALWAYS CORRECT.

To generate Question 1, we specify an assertion that essentially means "for any index k of lst, if k < i + 1, then lst[k] <= lst[i+1]". The tool inserts instrumentation code similar to the following at the point L1:

```
show([[i,j,lst]])              # (A)
show(alter(other,[i,j,lst]))  # (B)
```

Function show is also generated by the tool and contains code similar to the following:

```
for k in range(len(lst)):
  if k < i+1 and lst[k] > lst[i+1]:     # impossible
    .... print(k, i, lst, ...) ....
  elif k < i+1 and lst[k] <= lst[i+1]: # possible
    .... print(k, i, lst, ...) ....
  elif k >= i+1 and lst[k] < lst[i+1]: # possible
    .... print(k, i, lst, ...) ....
  ....
```

While function alter generates a list of states each of which is obtained by replacing the value of either i, j or an element of lst with the value of other.

The tool then runs PyExZ3 with the instrumented bubble sort function. Its dynamic symbolic execution mechanism runs the code symbolically, meaning that it tries to run every possible branch of each conditional and gives a concrete value for each variable when it meets the print function. Thus, instrumented code (A) shows reachable states from the branches marked as "possible". Code (B) shows unreachable states, due to the assertion, from the branch marked as "impossible".

3.3 Rubric 2-2

Rubric 2-2 indicates the level at which students are able to correct a given program that does not behave as intended. To evaluate whether students have achieved this level, we use incomplete or buggy code.

Our tool, therefore, generates expressions which, when used to replace original expressions at specified lines in the code, fail to satisfy some of the intended properties and, thus, the post-condition, a loop invariant, and/or the well-found nature of the measures.

4 Small Experiment

We conducted a small experiment of the proposed methods.

Twelve undergraduate students in School of Literature of a university participated in the experiment. They have learned programming languages, including Python, for a half to three years, but they have not spent much time in practicing programming. All of them have learned basic control structures of python (for and while statements) in a course, but none has taken courses of algorithms or data structures. Some of them have difficulties in comprehending nested loops.

Two functions written in Python were used. Question A asks about a function which counts even numbers included in a given integer list. The function used in Question B receives a list of positive integers and a positive integer. It removes occurrences of the integer from the list, moves the remaining integers forward (to smaller indices), and pads the emptied positions with zeroes.

All the questions correspond to Rubric 2-1. Question A has two sub-questions A-1 and A-2, while Question B has three, B-1, B-2 and B-3. Each sub-question has four to eight cases that the examinee answers either true or false. Sub-questions A-i and B-i are in the form of Question i in Section 3.2.

The examinees were given 30 minutes to answer the questions. After they completed, the authors conducted an interview for each examinee to evaluate their understanding of the source code.

The results are shown on Table 2. An ID from 01 to 12 is given to each examinee. The number of different cases for each sub-question is shown in parentheses under the sub-question number; there are six cases for A-1, for example. The table shows the number of correct answers of each examinee for each sub-question. The columns of "A-eval" and "B-eval" show the authors' evaluation of examinees' understanding for each function through the interview. Label "Y" indicates an examinee who showed good understanding and "N" is given to those who poorly understand the code.

From the results, some statistics are calculated as follows.

For the two groups "Y" and "N", the Mann-Whitney U value of the sum of A-1 and A-2 is 2.5 and the p-value (for two-sided) is 0.043. The U value of the sum of B-1, B-2 and B-3 is 5.0 and the p-value is 0.045.

Spearman's rank correlation coefficients of a few combinations of the scores of sub-questions are shown in Table 3. Although a couple of combinations show correlation to some extent, we cannot draw definite conclusions from the results due to the numbers of questions and examinees.

The interview revealed several issues on the experiment.

The text of questions contains explanation of the function. A number of students answered the questions, not based on their reading of the source code, but relying on the explaining text. This type of explanation should not be removed entirely because it would give the examinee an excessive burden, but we may need to investigate appropriate expression in the text.

Some students felt difficulties in grasping the state expressions. For example, State (1) of Question 1 in Section 3.2 consists of five equalities including two indices of a list. Although all information is needed, this long enumeration may distract the examinee.

In this experiment, we did not precisely define the criteria with which the examinees are divided into two groups. The grouping was done based on the authors' impression and could be biased by various factors. Moreover, the number of examinees is small. We plan to conduct another experiment with improved questions, grouping methods and a larger number of examinees.

Table 1. Scores of examinees

ID	A-1 (6)	A-2 (4)	A-eval	B-1 (6)	B-2 (6)	B-3 (8)	B-eval
01	1	2	Y	2	3	5	Y
02	6	2	Y	4	3	4	Y
03	6	2	Y	6	5	1	Y
04	5	3	Y	4	4	0	N
05	5	2	Y	6	4	5	Y
06	6	2	Y	4	0	0	N
07	2	3	N	0	0	0	N
08	5	3	Y	6	6	8	Y
09	3	2	N	2	2	6	N
10	5	4	Y	5	5	7	Y
11	0	3	N	2	5	4	N
12	4	2	Y	3	5	5	N

Table 2. Correlation coefficients

sub-questions	ρ	p-value
A-1 and A-2	0.25	0.43
B-1 and B-2	0.54	0.07
B-2 and B-3	0.50	0.09
B-1 and B-3	0.25	0.42

5 Concluding Remarks

As shown in previous sections, the tools that we have developed for Rubric 1-2, Rubric 2-1, and Rubric 2-2 can automatically generate questions to some extent. The tools can potentially aid the production of the large number of questions required for university entrance examinations. However, this requires human intervention at many stages. For example, additional constraints are necessary to avoid trivial questions because some incorrect states at a specified point can be immediately rejected by examining previous statements. Such constraints can be automatically generated by analyzing previous statements.

Since the tools can currently generate questions for only Rubrics 1-2, 2-1, and 2-2, it must be extended to generate questions for other rubrics. Rubric 1-1 seems relatively easy; we can insert grammatical errors to correct programs and ask students to detect them as we insert semantical errors for Rubric 2-2. For Rubric 3, methods developed for automatic assessment systems can be applied in the case of computer-based examination.

Finally, empirical evaluation of the quality of the questions generated, which extends the reported small experiment, is required. In fact, experimental testing of students using questions generated by our tool and evaluation of the effectiveness of our methods with a larger number of examinees are currently planned as part of the research project mentioned in Section 1 [4]. For such experiments, more questions must be generated from programs with correctness proofs.

Acknowledgment

This work was supported by Research on Evaluation Method in the Subject of Informatics for University Admission Examinations Based on Computational Approach supported by Ministry of Education, enPiT Pro Project supported by Ministry of Education, and JSPS KAKENHI Grant Number 16K00109.

References

1. MEXT, Curriculum guideline for high school, 2018 (in Japanese). http://www.mext.go.jp/a_menu/shotou/new-cs/1384661.htm
2. Masami Hagiya, Defining informatics across Bun-kei and Ri-kei, Journal of Information Processing, Vol.23, No.4, pp. 525-530, 2015. http://doi.org/10.2197/ipsjjip.23.525
3. Yasuichi Nakayama, et al., Current situation of teachers of informatics at high schools in Japan, Olympiads in Informatics, pp. 177-185, 2018.
4. Research and Development of Assessment Methods in Selection of Applicants to Universities in the subject "Informatics" from Approach Based on Informatics (in Japanese). http://www.uarp.ist.osaka-u.ac.jp/index.html
5. Kenji Matsunaga and Masami Hagiya, Characterization of thinking, judging and expressing in the rubrics of common subject "Information", The 10th Congress of National High School Informatics Education Study Group, 2017 (in Japanese).
6. Danijel Radošević, Tihomir Orehovački, and Zlatko Stapić, Automatic on-line generation of student's exercises in teaching programming, Central European Conference on Information and Intelligent Systems, CECIIS 2010, 7 pages, 2010.
7. Akiyoshi Wakatani and Toshiyuki Maeda, Evaluation of software education using auto-generated exercises, 2016 IEEE International Conference on Computational Science and Engineering, IEEE International Conference on Embedded and Ubiquitous Computing, and International Symposium on Distributed Computing and Applications to Business, Engineering and Science, pp. 732-735, 2016.
8. Christopher Douce, David Livingstone and James Orwell, Automatic test-based assessment of programming: A review, Journal on Educational Resources in Computing, 5(3), Article 4, 2005.
9. Thomas Ball and Jakub Daniel, Deconstructing Dynamic Symbolic Execution, Proceedings of the 2014 Marktober Summer School on Dependable Software Systems Engineering, 2015.
10. Nikolai Tillmann and Jonathan de Halleux, Pex - White box test generation for .NET, TAP 2008: Tests and Proofs, LNCS 4966, pp. 134-153, 2008.
11. Leonardo de Moura and Nikolaj Bjørner, Z3: An efficient SMT solver, TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems, LNCS 4963, pp. 337-340, 2008.
12. The Z3 Theorem Prover. https://github.com/Z3Prover/z3