Detecting Functional Differences using Automatic Test Generation for Automated Assessment in Programming Education

Ryoko Izuta*, Shinsuke Matsumoto*, Hiroshi Igaki†, Sachio Saiki‡, Naoki Fukuyasu† and Shinji Kusumoto*
* Osaka University, Osaka, Japan
Email: {r-izuta, shinsuke, kusumoto}@ist.osaka-u.ac.jp
† Osaka Institute of Technology, Osaka, Japan
Email: {hiroshi.igaki, naoki.fukuyasu}@oit.ac.jp
‡ Kochi University of Technology, Kochi, Japan
Email: saiki.sachio@kochi-tech.ac.jp

***Abstract*—Software testing is being leveraged in programming education for automated assessment of programming assignments. When using software testing in programming education, program specifications are provided as unit or integration tests, and students create programs that pass these tests. Although this method has various advantages, such as ensuring objective program specifications and automating the operation check, it also has many disadvantages. For example, detecting innovations, such as original specifications and functional extensions by an individual student, is difficult. The purpose of this research is to automatically detect functional differences among student programs in programming education using tests. In our proposed method, automatic test generation is applied to student programs, and the generated tests are mutually executed for other student programs. Furthermore, we classify the tests based on the execution path to obtain sets of tests that are capable of detecting functional differences.**

***Index Terms*—automated assessment, automated test generation, programming education, test execution path**

I. Introduction

Many researches propose using software testing in programming education (hereinafter referred to as test-based education) [1]–[4]. In test based education, the specifications of a class assignment can be explicitly defined as a unit test or an integration test (hereinafter referred to as pre-test). Providing pre-tests has various advantages to both students and instructors, such as ensuring objective task specifications and automating the task completion check. Test based education also has a high affinity with modern and practical software development such as test-driven development [5] and continuous integration [6]. In addition to the acquisition of programming skills, it is expected to gain practical experience in software development.

A problem with test-based education is that it is difficult to detect *integuity* made by individual students. *Integuity* is a functional difference between student source codes. This includes functional improvements and extensions that students implement in order to achieve a better grade. A typical example of *integuity* is the process of checking the validity of parameters, such as checking whether an object is null or checking the range of an integer value. Furthermore, original additional specifications and functional extensions that are not checked by the pre-tests are also considerable. So, *integuity* can be said as implementations outside the scope of pre-testing. We believe that teachers should detect *integuity*, give positive feedback and encourage their efforts, rather than forcing everyone to implement minimal functions.

However, it is fundamentally impossible to test all possible behaviors that exist [7]. Excessive testing, while effective in terms of software quality, is not an adequate strategy for education because it not only obscures the essence of the task specification, but also inhibits free thinking in programming. We believe that the implementation of humor that contributes to the enjoyment of programming, such as *easter eggs*, should be detected and praised in some way, rather than ruthlessly rejected by pre-tests. Moreover, other researches that focus on automatic assessment, such as AUTOGRADER [8], only assess functions that instructors expect and cannot detect *integuity*.

The goal of this research is the automatic detection of *integuity* among student source code in test-based education. In order to achieve this goal, we set the following challenges, as shown in Fig. 1.

- Challenge 1: How to generate tests with the ability to detect *integuity* and classify these tests.
- Challenge 2: How to automatically detect *integuity* using the generated tests.

In Challenge 1, we generate tests from each source code developed by students that have the ability to detect *integuity*, which cannot be achieved by the pre-tests alone. In Challenge 2, the resulting tests are applied to each source code to detect the presence of *integuity*.

In this paper, we propose a test clustering method that uses automatic test generation and execution path information to achieve Challenge 1. In this method, we first apply automatic test generation to all submitted source code to obtain tests with the ability to detect *integuity*. Furthermore, every generated test is applied to all of the students, and the execution paths are

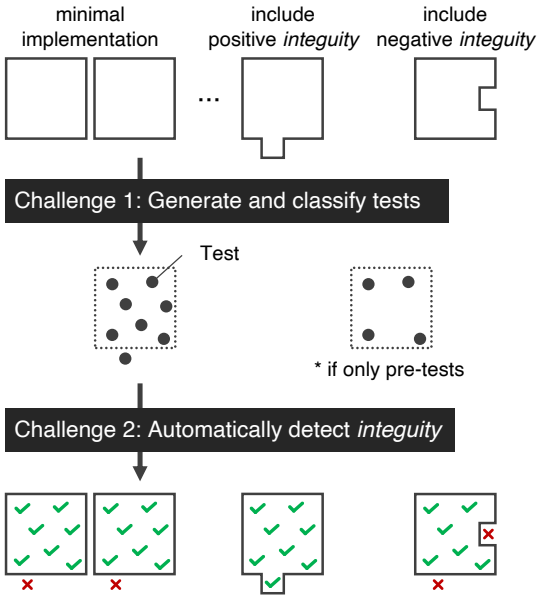

Fig. 1: Two challenges in automatic detection of functional differences

measured in order to extract the steps to verify the behavior of each test. Finally, by vectorizing and clustering the paths of each test, we identify the tests that have the same verification procedure. In this paper, we also report the results of a preliminary experiment on a programming subject having a few lines of code.

II. Motivating Example

In this section, we explain the motivation of this research using simple java source code. The subject is the PortNumber class, which represents port numbers in the TCP/IP protocol. An example code of this class and a pre-test for the constructor are shown in Fig. 2. This PortNumber class consists of the following elements: an int type field `number` (line 2) representing the port number, the constructor (line 3), a getter method for `number` (line 4), and a stringification method for `number` (lines 5 through 8).

As the first implementation task of this class, students are told to write their own PortNumber class constructor that passes the pre-test shown in Fig. 2(b). Two possible answers are shown in Fig. 3. The constructor of Student A (Constructor A) in Fig. 3(a) is exactly the same as the example constructor and implements only the minimum functionality that passes the pre-test. On the other hand, Constructor B in Fig. 3(b) implements a validation check of the port number and does not accept negative values.

Constructor B is an implementation that contains positive features not included in the example, which should be detected and graded appropriately. However, the pre-test only checks whether the constructor implements the minimal function correctly, and does not have the ability to check additional functions. Although visual confirmation is an option, it is not realistic considering the number of students in programming

```
1  public PortNumber {
2    int number;
3    public PortNumber(int n) {number = n;}
4    public getNumber() {return number;}
5    public toString() {
6      if (number == 22) return "ssh";
7      if (number == 80) return "http";
8      ... }
9  }
```

(a) Example code for PortNumber class

```
1  @Test public void test1() {
2    int number = new PortNumber(22).getNumber();
3    assertEquals(22, number);
4  }
```

(b) Pre-test for PortNumber constructor

Fig. 2: Example code and pre-test for PortNumber class

```
1  public PortNumber(int n) {
2    number = n;
3  }
```

(a) Constructor of Student A

```
1  public PortNumber(int n) {
2    if (n < 0) number = 0; // originality
3    else number = n;
4  }
```

(b) Constructor of Student B

Fig. 3: Student constructor examples

courses, such as university courses which could include 10 to 100 students.

III. Proposed Method

A. Overview

In order to detect *integuity* in the source code of multiple students, we use automatic test generation. First, tests are generated automatically from each student source code. Then, each test is applied to every student, not just the base student used for test generation, to identify functional differences.

Simply applying automated test generation will result in numerous tests. These tests most likely include tests that can detect functional differences. However, many redundant tests are also included, such as tests that only check the same functionality as the pre-tests. Visual inspection is necessary in order to understand the "meaning" of a test and the functional difference that is represented by that test. In order to reduce the effort required, appropriate classification of the generated tests is needed.

Therefore, the proposed method is divided into three major steps, as follows: (1) automatic test generation to generate tests with the ability to detect *integuity*, (2) classification of the generated tests, and (3) visual inspection of the functional differences. In this paper, we explain Steps 1 and 2. Step 3 is related to Challenge 2 and is left for a future study.

B. Step 1. Automated Test Generation

First, we apply automatic test generation to every student's source code and generate unit tests. There are many studies on automatic test generation [9], and open-source tools, such as EvoSuite [10] and Randoop [11] , have been released. Automatic test generation uses exploratory meta-algorithms, such as genetic algorithms, and execution path analysis to generate tests that maximize coverage.

Functional differences between source codes often appear as differences in branches, i.e., execution paths. Therefore, tests with the ability to detect *integuity* can be obtained by automated test generation, which aims to maximize coverage.

C. Step 2. Test Classification

In Step 2, we first mutually apply the tests generated in Step 1 to every student's source code. If a total of M tests are generated from N source codes, then $N \times M$ test executions are performed. In addition, for each test execution, the execution path is recorded. The mutual execution of tests and the recording of path information are the preprocessing steps for the following test classification.

Here, we consider the criteria that should be used to calculate the similarity of tests in the test classification phase. The goal of this research is to automatically detect *integuity*. Therefore, the similarity should be based on the function that each test is testing.

An option is to use the state of each test execution (pass/-fail). However, test states are not suitable for classifying these tests because test states are automatically generated. Since the true oracle of the tests are unknown [12], automated test generation assumes that the base source code behaves correctly. As a result, there is excessive variety in the test assertions, and some assertions might even be in contradiction.

We decompose a test into two parts: the execution part, which represents the procedure of the test, and the assertion part, which checks whether the execution result matches the expected value. The execution path of only the execution part best represents what function each test is testing. Therefore, we record the execution path and use it to classify the generated tests. Unlike other researches about test prioritization [13]–[15], by using the path information, classification is possible from the viewpoint of the check procedure of each test, leaving out the assertion part.

In order to use the path information, we calculate a vector for each test. The component of each vector is a hash of a serialization of the student ID of the class executed and the execution path. For example, if student source code x is tested and the execution path is 11101 (meaning that only the fourth line out of all five lines was not passed), then string $x11101$ is generated and hashed. Note that each vector component is a hash value and cannot be treated as a numeral.

Finally, the tests are classified based on the aforementioned vectors. Currently, tests that have the exact same vectors are classified as belonging to the same set. In this way, tests that have the same execution paths, i.e., tests with the same functions, will be classified together. This test classification method can be fully automated even for source codes with 100 participants, and can classify a large number of generated tests in terms of the similarity of their verification procedures.

TABLE I: Subject of the experiment

Name of subject	When `n` < 0	When `n` > 65535
CheckNone	substitute `n`	substitute `n`
CheckLow-SetAltV	substitute 0 [*a]	substitute `n`
CheckLow-ThrowE	throw exception[*a]	substitute `n`
CheckUpp-SetAltV	substitute `n`	substitute 0[*b]
CheckUpp-ThrowE	substitute `n`	throw exception[*b]
CheckBoth-SetAltV	substitute 0[*a]	substitute 0[*b]
CheckBoth-ThrowE	throw exception[*a]	throw exception[*b]

[*a]*Integuity* 1 (lower bound check), [*b]*Integuity* 2 (upper bound check)

IV. Preliminary experiments

We applied our approach to a programming subject having a few lines of code in order to confirm whether our approach can generate tests with the ability to detect *integuity* and classify them appropriately.

A. Subjects

The subject used in the experiments is the PortNumber class as described in Section II. The assumed *integuity* consists of a combination of two factors. One is whether or not to check the range of the port number, and the other is the behavior in case of an error. There are three types of range confirmation: lower limit confirmation (`n < 0`), upper limit confirmation (`n > 65535`), and both lower limit confirmation and upper limit confirmation. There are two types of behavior when the specified port number is out of range: assigning an alternate value (zero) and raising an exception.

As a result of the combination of the above two factors, we have created a subject with seven functional differences as shown in Table I. The names of the subjects are a combination of range checking and error handling. For example, CheckNone does not perform any range checking and assigns the specified value as it is. This is the minimum implementation of the issue requirements and corresponds to the constructor shown in Fig. 3(a). In CheckLow-SetAltV, if the value is below the lower limit, then an alternative value is assigned. No upper limit check is performed. This is equivalent to the constructor shown in Fig. 3(b).

The *integuity* assumed in this preliminary experiment is whether the two types of range checking processes are performed, and the behavior in case of an error should be ignored. For example, the bottom four lines of Table I (CheckUpp-* and CheckBoth-*) should all be detected as having the same functional difference (checking the upper bound), although they behave differently in case of an anomaly. For the definition of this functional difference, please refer to Section III-C.

B. Experimental Method

First, we manually created seven types of subject source code as described in Section IV-A. Second, we used EvoSuite,

Student source / Test	ChkNone	ChkLow-SetAltV	ChkLow-ThrowE	ChkUpp-SetAltV	ChkUpp-ThrowE	ChkBoth-SetAltV	ChkBoth-ThrowE	Set ID
ChkNone1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone3	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE2	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkLow-ThrowE3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkUpp-SetAltV4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkUpp-ThrowE4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkBoth-SetAltV4	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkBoth-SetAltV5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkBoth-ThrowE4	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkBoth-ThrowE5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
# generated tests	3	3	3	4	4	5	5	
# path types	2	4	3	4	3	5	3	

Fig. 4: List of generated tests and runtime paths

```
1  @Test public void CheckNone2() {
2    PortNumber portNumber0 = new PortNumber(992);
3    int int0 = portNumber0.getNumber();
4    assertEquals(992, int0);
5  }
```

(a) Example of set ID1

```
1  @Test public void CheckNone3() {
2    PortNumber portNumber0 = new PortNumber(-1);
3    int int0 = portNumber0.getNumber();
4    assertEquals(-1, int0);
5  }
```

(b) Example of set ID2

```
1  @Test public void CheckLowThrowException2() {
2    PortNumber portNumber0 = null;
3    try {
4      portNumber0 = new PortNumber(-394);
5      fail("Expecting␣exception:␣
           IllegalArgumentException");
6    } catch(IllegalArgumentException e) {
7     verifyException("PortNumber", e);
8    }
9  }
```

(c) Example of set ID3

Fig. 5: Examples of sets ID1 through ID3

an automatic test generation tool, with the default parameters to generate tests for the subject source code. Third, we ran every test for every source code along with JaCoCo[1], an execution path measurement tool, to obtain path information during the test execution. Finally, we calculated the vector for each test with the method explained in Section III-C. The `md5sum` command was used for hashing.

C. Results and Discussion

The hashes for each test obtained from the experiment are shown in Fig. 4. Each column represents a source code, and each row represents a test method. Each value represents the path hash for the execution. Each test vector consists of the hashes on its right. In order to improve visibility, the same hash in each column is filled with the same color. The name of each test method is represented with the name of the source code from which the test was generated followed by a sequential number. Note that the order of the tests in the vertical direction is based on the order generated by EvoSuite and has no meaning. The set IDs in the right-hand column of the table are the classification results based on the exact matching of the vectors. Furthermore, the meaning of each set is shown in Table II, and examples of a specific generated test for sets ID1 to ID3 are shown in Fig. 5.

As a result of clustering, most of the tests were assigned to set ID1. There is a variety of input values `n`, but we can see from the test meaning in Table II and the test example in Fig. 5(a) that all of the tests in set ID1 have inputs in the normal range ($0 \leq n \leq 65535$). These set ID1 tests are generated from all of the source codes and can be interpreted as a shared function.

Tests in set ID2, such as Fig. 5(b), are tests that check what happens with a negative input. Both tests that expect the negative value itself and tests that expect an alternative value zero were classified as belonging in set ID2. This shows that tests checking the lower bound are properly classified to the same set without depending on the assertion part. All tests in set ID4 likewise check the upper bound.

On the other hand, sets ID3 and ID5 were sets overly subdivided. Set ID3 is identical to set ID2 in terms of testing $n \leq 0$, and ID5 is identical to ID4 in terms of testing $n \geq 65535$. These sets should be classified as the same set. This excessive division occurs because of unexpected path deviation. As shown in Fig. 5(c), all tests expect thrown executions and lack the call to the getter method. The presence or absence of the getter call affects the execution path, which results in a difference in the path vectors and the classification. Indeed, CheckNone takes two different paths, `765d` and `258a`, as shown in Fig. 4 even though CheckNone implements only the minimal function. Tests with `765d` include getter calls, while tests with `258a` do not. Only the tests with `258a` belong to sets ID3 and ID5. Tests that expect thrown exceptions tend to be overly subdivided, and this needs to be addressed.

V. Conclusion

We proposed a test classification method that uses automatic test generation and execution path information to detect *integuity* among student source code.

In the future, we plan to address the following issues. First, it is essential to deal with the excessive test segmentation

[1]https://www.jacoco.org/jacoco/

TABLE II: Set ID of the test and its meaning

set ID	Range of n to substitute	expectation
1	$0 \leq \mathtt{n} \leq 65535$	n
2	$\mathtt{n} < 0$	n or alternative value 0
3	$\mathtt{n} < 0$	Exception
4	$65535 < \mathtt{n}$	n or alternative value 0
5	$65535 < \mathtt{n}$	Exception

that occurred in the preliminary experiment. In addition, the automatic detection of *integuity* using generative tests, which we mentioned as Challenge 2, is an important remaining task. In order to achieve Challenge 2, we are considering clustering based on test state in addition to the current clustering, and analyzing the characteristics of each group (number of test passes, number of tests in the same group, etc.) that emerge from the results. The application of the system to educational settings is also an important issue, and we are considering introducing the system to the programming exercises that we are conducting.

ACKNOWLEDGMENT

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: JP21H04877, JP17K00500, JP19K02973, JP19K03001, and JP21K11829)

REFERENCES

[1] F. Restrepo-Calle, J. J. Ramírez Echeverry, and F. A. González, "Continuous assessment in a computer programming course supported by a software tool," *Computer Applications in Engineering Education*, vol. 27, no. 1, pp. 80–89, 2019.

[2] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proc. Koli Calling International Conference on Computing Education Research*, 2010, pp. 86–93.

[3] S. H. Edwards and M. A. Pérez-Quiñones, "Experiences using test-driven development with an automated grader," *Journal of Computing Sciences in Colleges*, vol. 22, no. 3, pp. 44–50, 2007.

[4] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, vol. 5, no. 3, pp. 4—-es, 2005.

[5] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *SIGCSE Bulletin*, vol. 40, no. 2, pp. 97–101, 2008.

[6] B. P. Eddy, N. Wilde, N. A. Cooper, B. Mishra, V. S. Gamboa, K. M. Shah, A. M. Deleon, and N. A. Shields, "A pilot study on introducing continuous integration and delivery into undergraduate software engineering courses," in *Proc. Conference on Software Engineering Education and Training*, 2017, pp. 47–56.

[7] C. Kaner, C. Cem, and K. All, "The impossibility of complete testing," 1997.

[8] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: An approach based on formal semantics," in *Proc. International Conference on Software Engineering: Software Engineering Education and Training*, 2019, pp. 126–137.

[9] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[10] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proc. ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, 2011, pp. 416–419.

[11] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, 2007, pp. 815–816.

[12] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *Proc. International Conference on Software Testing, Verification and Validation*, 2013, pp. 342–351.

[13] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[14] J. Chen, L. Zhu, T. Y. Chen, D. Towey, F.-C. Kuo, R. Huang, and Y. Guo, "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *Journal of Systems and Software*, vol. 135, pp. 107–125, 2018.

[15] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *Proc. IEEE International Conference on Software Maintenance*, 2011, pp. 382—-391.