Int. J. Advance Soft Compu. Appl, Vol. 5, No. 3, December 2013
ISSN 2074-8523; Copyright © SCRG Publication, 2013

Designing a Test Set for Structural Testing in Automatic Programming Assessment

Rohaida Romli[1], Shahida Sulaiman[2], and Kamal Zuhairi Zamli[3]

[1]School of Computing
College of Arts and Sciences
Universiti Utara Malaysia, 06010 UUM Sintok, Kedah, Malaysia.
e-mail: aida@uum.edu.my

[2]Faculty of Computing,
Universiti Teknologi Malaysia, *81310 UTM Skudai, Johor, Malaysia.*
e-mail: shahidasulaiman@utm.my

[3]Faculty of Computer Systems and Software Engineering,
Universiti Malaysia Pahang, Lebuhraya Tun Razak, 26300 Gambang, Kuantan, Pahang, Malaysia.
e-mail: kamalz@ump.edu.my

Abstract

An automatic programming assessment (APA) method aims to support marking and grading of students' programming exercises. APA requires a test data generation to perform a dynamic testing on students' programs. In software testing field, diverse automated methods for test data generation are proposed. Unfortunately, APA seldom adopts these methods. Merely limited studies have attempted to integrate APA and test data generation to include more useful features and to provide a precise and thorough quality of program testing coverage. Thus, we propose a test data generation approach to cover both the functional and structural testing of a program for APA by focusing the structural testing in this paper. We design a test set based on the integration of positive and negative testing criteria that enhanced path coverage criterion to select the desired test data. It supports lecturers of programming courses to furnish an adequate set of test data to assess students' programming solutions in term of structural testing without necessarily having the expertise in a particular knowledge of test cases. The findings from the experiment depict that the test set improves the criteria of reliability and validity for test data adequacy in programming assessments.

Keywords: *Automatic Programming Assessment (APA), test data generation, structural testing, path coverage, positive testing, negative testing*

1 Introduction

Automatic Programming Assessment (APA) is commonly known as an alternative method to automatically mark and grade students' programming solutions. It aims to overcome the manual assessment, which is time-consuming and requires much effort and attention that are prone to error in any levels of assessment [1]. Besides, APA offers important benefits in terms of immediate feedback, objectivity and consistency of the evaluation as well as a substantial saving of time in the evaluation of the assignments [2] without the need to reduce exercises [3]. Due to huge class sizes in Computer Science or IT programme which might have hundreds of students in a single course [4], the practice of programming exercises assessment leads to extensive workload to lecturers or instructors particularly if it has to be carried out manually. This typical approach of assessment may lead to unintended biases and different standard of marking schemes. Furthermore, the feedback provided to students through marking is generally limited, and often late and outdated particularly to the topic dealt in the assignment [5].

To date, a number of automatic tools available for APA such as Assyst [6], BOSS [7], GAME [8], TRAKLA2 [9], PASS [10], ELP [11], CourseMaster [12], WeBWork [13], SAC [14], Oto [15], ICAS [16], PETCHA [17], eGrader [18], and Bottlenose [19]. Such tools provide advantages not only to lecturers, but may also play an important role in students' learning outcomes [20]. Typically, dynamic correctness assessment of students' programs involves the process of program quality testing through the execution of program with a range of test data and monitoring it conformance through the comparison between the outputs produced and the expected ones [21]. Thus, there is a need to prepare an appropriate set of test data that conformance to program specifications and to detect an occurrence of errors or unexpected mistakes in students' programs.

In software testing research area, various of studies propose automated methods for test data generation [22][23][24][25][26][27][28][29][30]. Despite the potentials of the proposed methods in providing the most efficient way to generate test data for large-scale projects, researches in APA seldom adopt these methods. To date, very limited studies have attempted to incorporate both automation of test data generation and programming assessment [9][31][32][33]. We intend to enhance the previous studies as the methods proposed are merely applied as a simple technique to generate test data, or they derive test data based on only the functional or structural aspect of a program separately. Other than that, the studies such as proposed by Guo *et al.* [34] and Cheng *et al.* [35] seem to apply the external tools (legacy systems) to generate test data for functional or structural testing. However, the accessibility of such tools might be a problematic. Some other works utilize the JUnit framework to design test cases [5][36][37][38][14][15][39] or use specific technique (in manual way) [40][41]. However, JUnit requires high technical skill to code particular run tests that requires rubric in mind. Hence, this limiting its use for advanced users. Therefore,

it motivates us to propose a test data generation approach for the dynamic functional and structural testing of a program for APA so-called FaSTDG (**F**unctio**a**l-**S**tructural **T**est **D**ata **G**eneration) approach. However, this paper will only highlight on the part of structural testing. Section 4 will review the related work particularly on this testing part that justifies the issues and challenges, which reveals among the possible gaps to be explored.

This paper consists of six sections. Section 2 details up the design of test set for test data selection based on an improved path coverage criterion. In the following section, it demonstrates how the test set allocates an adequate set of test data based on a sample of programming exercise. Section 4 reveals the analysis and findings from the conducted controlled experiment to evaluate the completeness coverage of structural testing as a part of FaSTDG approach. Section 5 reviews the related work with regard to test data generation for structural testing in APA. Finally, Section 6 concludes the paper.

2 Path Coverage Criterion to Design a Test Set

This study adapts path coverage testing criterion as the criteria are used to examine the correctness of structural testing of students' program. In terms of the adequacy criterion used in corresponding to path coverage testing, this study employs control-flow test adequacy criterion to guide in finding the desired paths. Instead of considering all finite paths from start to end (positive testing or test adequacy criteria-reliability), this study also incorporates negative testing criteria (or test adequacy criteria-validity) that is based on error-based adequacy criteria which requires test data to check programs on certain error-prone points [42]. In order to obtain feasible paths, this study adapts the technique of boundary-interior path testing. The notion of criteria of test data selection for this study is available from our previous work [43]. As this study incorporates positive and negative testing criteria, inclusion to valid (true) and invalid (false) path conditions, the study also embeds an illegal path condition in designing test cases.

For structural test data generation, the process of deriving test data depends on two main control structures with regard to the flow of control [44][45], which are selection control structure, loop control structure, and combination of the both.

2.1 Selection control structure

Focusing Java programming, selection control structures concern *if*, *if…else* and *switch…case* decision-making statements [46]. All the statements may involve consecutive (sequential) or nested structure. This study considers the both structures. The detail of test cases designed for each decision-making structure is as follows:

2.1.1 Selection-Consecutive (Sequential)

For this type of decision-making structure, the means of deriving test cases relies on the following properties:

(i) The number of selection control structures a program has,
(ii) The number of options or decisions each selection control structure has, and
(iii) The category of input conditions (valid, invalid, illegal) that a lecturer prefers to consider.

For all cases, by default, the derived test cases involve valid input conditions. Invalid and illegal input conditions may be included based on a lecturer's preference. For example (Case 1), if a tested program has two selection control structures (*Selection1* and *Selection2*), and then *Selection1* and *Selection2* has two and three options respectively. Considering that, a lecturer prefers to include all input conditions, Table 1 shows the generated test cases with their respective input conditions.

Based on Table 1, the total number of generated test cases is nine. For *Selection1*, in a total it contains four test cases (TC1, TC2, TC3, and TC4). The reason is *Selection1* has two options (representing valid input conditions) with another two cases (representing invalid and illegal input conditions). In addition, *Selection2* comprises of five test cases as it has three options representing valid input conditions a long with another two test cases, which represent invalid and illegal input conditions. The pairs of (TC3, TC8) and/or (TC4, TC9) will be excluded if a lecturer do not prefer to include invalid and/or valid condition (s).

Table 1: Generated test cases based on the example of Case 1 for Selection-Consecutive (sequential)

Test Case	Input Conditions
TC1	Valid-Option1-Selection1
TC2	Valid-Option2-Selection1
TC3	Invalid-Selection1
TC4	Illegal-Selection1
TC5	Valid-Option1-Selection2
TC6	Valid-Option2-Selection2
TC7	Valid-Option3-Selection2
TC8	Invalid-Selection2
TC9	Illegal-Selection2

2.1.2 Selection-Nested

The parameters considered in deriving test cases for this type of decision-making structure is identical to as explained in 2.1.1. However, this type of decision-making structure will consider a combination of input conditions among the selection control structures that a tested program has. A tree structure can visualize this decision-making structure that focuses only valid input conditions

among all the selections (*Selection1*, *Selection2*, …, *SelectionN*). The number of parent nodes in the tree represents the number of selections that the tested program has. The number of children for each parent node denotes the number of options for the respective selection. If the selection has another inner selection, then the first child node of the outer selection becomes a parent node for the inner selection.

For example, if a decision-making structure of *if…else* statement is given as shown in Fig. 1, the respective test cases generated is shown in Table 2.

```
if (booleanExpression-Option1) → Selection1
      if (booleanExpression-Option1) → Selection2
            if (booleanExpression-Option1) → Selection3
                  .
                  .
            else → Option2    → Selection3
            .
            .
      else  → Option2   → Selection2
            .
            .
else if (booleanExpression-Option2) → Selection1
.
.
else  → Option3 → Selection1
.
.
```

Fig. 1: Example of decision-making statement for Selection-Nested

Table 2: Generated test cases for Selection-Nested based on Fig. 1 (considering all input conditions)

Test Case	Input Conditions
TC1	Valid-Option1-Selection1, Valid-Option1-Selection2, Valid-Option1-Selection3
TC2	Valid-Option1-Selection1, Valid-Option1-Selection2, Valid-Option2-Selection3
TC3	Valid-Option1-Selection1, Valid-Option2-Selection2
TC4	Valid-Option2-Selection1
TC5	Valid-Option3-Selection1
TC6	Invalid- Selection1, Invalid-Selection2, Invalid-Selection3
TC7	Illegal- Selection1, Illegal-Selection2, Illegal-Selection3

Based on Table 2, the number of generated test cases for valid input conditions is five (TC1 to TC5) as the number of the trees' leaves are five. However, for invalid and illegal input conditions, this study only includes one test case to represent each input condition (TC5 and TC7). The main reason is to

reduce the total number of test cases generated. Besides, there have already been included the test cases to exercise valid input conditions hence, no use to include the test cases that has a combination of valid and invalid input conditions.

2.2 Loop Control Structure

In Java programming, there are three types of loop statements: *while* loops, *for* loops and *do-while* loops [46]. For all the loops, instead of controlling the loop based on a specific number of repetitions (counter value), another common technique is to designate a special value to control repetitions (sentinel value). The *for* loop commonly gets involved with counter value.

This study categorizes the types of loops based on the repetitions are controlled by either counter or sentinel value. The terms use to represent both types are CounterLoop and SentinelLoop. The detail test cases design for both types of loops are as follows:

2.2.1 CounterLoop-Consecutive (sequential)

The design of test cases for this kind of loop structure depends on the number of loops that a program has, and input conditions that a lecturer wishes to consider. In order to ensure a finite number of paths tested for the loop, this study employs the modified boundary-interior approach. Commonly, the approach tests a loop to be entered but not iterated [47]. However, this study allows a lecturer to specify the number of iterations used to test the loop. This concept applies to both valid and invalid input conditions of the loop.

For example (Case 2), if a tested program has three loops (*Loop1*, *Loop2*, and *Loop3*), and a lecturer prefers to include all input conditions (valid, invalid and illegal) in the design of test cases, Table 3 shows the generated test cases with their respective input conditions.

Table 3: Generated test cases for Counter Loop–Consecutive (sequential) based on the example of Case2

Test Case	Input Conditions
TC1	Valid-Loop1
TC2	Invalid-Loop1
TC3	Illegal-Loop1
TC4	Valid-Loop2
TC5	Invalid-Loop2
TC6	Illegal-Loop2
TC7	Valid-Loop3
TC8	Invalid-Loop3
TC9	Illegal-Loop3

Based on the table, the total number of generated test cases is nine. For all loops, each of them has three test cases representing valid, invalid and illegal

input conditions of the loop. If a lecturer intends to exclude invalid and/or illegal input condition(s), the combinations of (TC2, TC5, TC8) and/or (TC3, TC6, TC9) will not in the list of generated test cases.

2.2.2 CounterLoop-Nested

The design of test cases for CounterLoop–Nested is similar to CounterLoop–Consecutive (sequential), except the way to generate test cases making a consideration of combining input conditions among the loops contained in a program. The combination technique ignores any duplicate combinations among input conditions of the considered loops. These input conditions can be any two combination of valid-invalid-illegal, valid-invalid or valid-illegal input conditions. The remaining input conditions of certain loops will only be combined with input conditions of other loops that have not been combined yet.

This technique is proposed to mainly reduce the number of test cases generated. In addition, it is able to cover an adequate combination of input conditions by assuming that as long as if one input condition for the respective loop has already been covered in the combination, there is no use to consider it again in the next combination. The reason is that once a testing executes both valid input conditions of the inner and outer loops, hence there is no use to re-execute the testing on the same valid input condition of the inner loop but with the other input conditions of its outer loop. For example, a combination of invalid-valid or illegal-valid input condition, as by the fact always produces invalid or illegal results. Therefore, it is more significant to create testing on the combination of invalid-invalid, invalid-illegal and illegal-illegal input conditions to produce the possibility of invalid and illegal results. In addition, structural testing of this study mainly emphasizes the adequacy of path coverage criterion, hence the exhaustive combination (n^k) of input conditions is not too crucial.

Fig. 2 illustrates the combination technique. Assuming that a program has three loops (*Loop1*, *Loop2*, *Loop3*) and each of them considers valid, invalid and illegal input conditions (Case 3). Table 4 depicts the test cases generated for the program. Based on the table, they are ten test cases generated as the tested program has three loops. For this type of loop, as the loops are arranged in a nested structure, hence the number of test data involved in each test case is as equation (1).

$$\textit{Number of test data in each test case} = \prod_{i=1}^{n} \textit{number of repeatitions}$$

where, $i = 1, 2, \ldots\ldots, n$

n – number of loops (1)

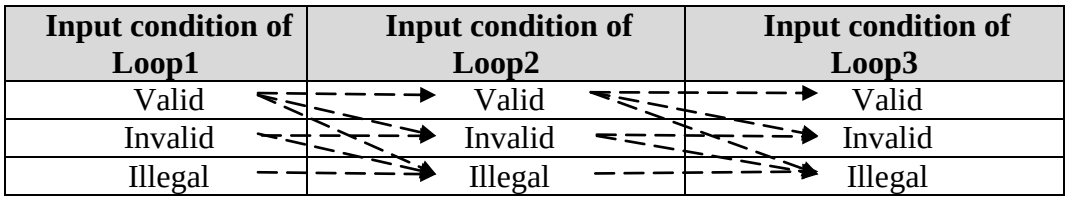

Fig. 2: Combination technique among input conditions of three loops

Table 4: Generated test cases based on an example in Fig. 2 for Counter Loop–Nested (Case 3)

Test Case	Input Conditions
TC1	Valid-Loop1, Valid-Loop2, Valid-Loop3
TC2	Valid-Loop1, Valid-Loop2, Invalid-Loop3
TC3	Valid-Loop1, Valid-Loop2, Illegal-Loop3
TC4	Valid-Loop1, Invalid-Loop2, Invalid-Loop3
TC5	Valid-Loop1, Invalid-Loop2, Illegal-Loop3
TC6	Valid-Loop1, Illegal-Loop2, Illegal-Loop3
TC7	Invalid-Loop1, Invalid-Loop2, Invalid-Loop3
TC8	Invalid-Loop1, Invalid-Loop2, Illegal-Loop3
TC9	Invalid-Loop1, Illegal-Loop2, Illegal-Loop3
TC10	Illegal-Loop1, Illegal-Loop2, Illegal-Loop3

2.2.3 SentinelLoop-Consecutive (sequential)

The design of test cases The way to derive test cases for SentinelLoop–Consecutive (sequential) structure is identical to the CounterLoop–Consecutive (sequential). Its minor dissimilarity is merely in terms of the means of configuring the loop parameters (sentinel values). For CounterLoop the configuration involves counter loop values to determine how many times a certain loop iterates. In order to ensure a finite number of paths are tested for the sentinel loop, this study also employs the boundary-interior approach. For this type of structure, test cases are designed to test the loop to be entered but not iterated. This differs as compared to CounterLoop in which a lecturer can decide to test a certain number of repetitions for each single loop.

The means of deriving test cases are exactly the same as CounterLoop–Consecutive (sequential). In terms of the number of test data involved in each test case, as the loop does not involve the repetition concept, the test data follow exactly the number of input variables or parameters that the tested program has.

2.2.4 SentinelLoop-Nested

This kind of loop structure applies the combination of concepts CounterLoop-Nested and SentinelLoop-Consecutive (sequential).

2.3 Combination of Selection and Loop Control Structures

Similar to the other two structures as previously discussed (sub-sections 2.1 and 2.2), this part also considers both consecutive and nested design structures. The detail design of test cases of both types are as below:

2.3.1 CounterLoop and Selection (consecutive/sequential)

The test cases design for this design structure is a combination of Selection–Consecutive (sequential) and CounterLoop–Consecutive (sequential).

2.3.2 CounterLoop and Selection (nested)

There are two possibilities of arranging this type of program structure: Counter loop occurs as an outer part of selection control structure (CounterLoop → Selection), or selection control structure is an outer structure of counter loop (Selection → Counter-Loop). The way to derive test cases for both types of arrangements is rather different as both counter loops and selection control structures are collectively different with each other in terms of flow of control. The following describes the test cases designed of both arrangement types:

A. Counter-Loop → Selection

Each of the selection control structures defined has a possibility to occur in any counter loops consisted in the tested program. However, every counter loop will be arranged at their level of nested sequentially based on given naming conventions.

For example (Case 4), assuming that a tested program P1 has two counter loops and one selection control structure with two options and it is part of the second loop. The respective naming conventions for the loops are *Counter-Loop1*, and *Counter-Loop2*. *Selection1* and *Selection2* consisted in *Counter-Loop2*. The limitation here is, it is not applicable if one or more counter loops defined in the tested program is arranged consecutively. Suppose its code structure will be viewed as in Fig. 3. If all the three input conditions are taken into consideration in deriving test cases, Table 5 tabulates the corresponding test cases.

The way to derive test cases does not rely on a combination of Selection-Nested and CounterLoop-Nested. As shown in Table 5, *Valid-Loop1* represents test case for valid input condition of *Loop1*. It is optional either to consider an input condition of *Loop2* as valid, invalid or illegal. A lecturer can decide to choose one of them as long as the control flow enters *Loop1*. However, the test cases from TC4 until TC8 will test on paths that enter *Loop2*. Thus, these test cases definitely are valid input condition of *Loop1*. The same concept applies to TC2 and TC3 as well as TC8 and TC9. However, for TC8 and TC9 it is possibly to select any type of input condition (valid, invalid or illegal) of *Loop1*. By applying this concept, it is probably able to deduce the total number of generated test cases. In addition, it does cover the minimal coverage paths.

```
Counter-Loop1 {
      //body of Counter-Loop1
      Counter-Loop2 {
            //body Counter-Loop2
            Option1-Selection1
            {     //body Option2-Selection2     }
            Option2-Selection1
            {     //body Option2-Selection2     }
      }
}
```

Fig. 3: Example of code structure for Counter-Loop → Selection (Case 4)

Table 5: Generated test cases based on an example of Case 4 for Counter-Loop → Selection

Test Case	Input Conditions
TC1	Valid-Loop1
TC2	Invalid-Loop1
TC3	Illegal-Loop1
TC4	Valid-Loop2, Valid-Option1-Selection1
TC5	Valid-Loop2, Valid-Option2-Selection1
TC6	Valid-Loop2, InvalidOption-Selection1
TC7	Valid-Loop2, IllegalOption-Selection1
TC8	Invalid-Loop2
TC9	Illegal-Loop2

B. Selection → Counter-Loop

The means of deriving test cases and the sequence among the control structures involved are rather different in certain aspects compared to CounterLoop → Selection. Each counter loop has a possibility to occur in any selection control structures. However, among the selection control structures, this study only takes into consideration if they are arranged in a consecutive way. Even in a programming practice, there have been a number of possibilities of design structures that involve selection control structure and counter loop, but the main focus here is to ensure the nested structure only applies between the selection control structure and counter loop. In addition, in a case of the elementary programming course, commonly programming assignments that involve this control structures do not happen to have too complicated design structure to support the applicable topic in a course syllabus.

For each selection control structure, any counter loops that occur in it, they will be arranged at the level of nested sequentially based on given naming conventions. Such arrangement is identical to CounterLoop → Selection. For example (Case 5), assume that a tested program P2 has two selection control structures (*Selection1* and *Selection2*) and counter loops (*Counter-Loop1* and

Counter-Loop2) respectively. It is given that *Selection1* has a single option and *Selection2* otherwise has two options. Suppose its code structure is shown in Fig. 4. In this case, it is not applicable if for a certain selection control structure there exist two or more counter loops that are arranged consecutively.

If all the three input conditions are taken into consideration in deriving test cases, Table 6 shows their corresponding test cases. The way of deriving test cases is quite similar to CounterLoop → Selection. However, as an outer structure starts with the selection control structure, the test will evaluate the outer prior the inner (counter loop) structures. As shown in Table 6, TC1 until TC3 exercise valid input condition of *Selection1* and valid, invalid or illegal input condition of *Loop*1. Then, TC4 and TC5 follow the test to exercise the invalid and illegal input conditions of *Selection1* respectively. The same thing goes to *Selection2* and *Loop2* in which their respective test cases are TC6 until TC11. As *Selection2* has two options, a test case that exercises valid input condition of the second option should occur (TC9).

```
Option1-Selection1 {
      //body of Option1-Selection1
      Counter-Loop1
            {//body of Counter-Loop1 }
}
Option1-Selection2 {
      //body of Option1-Selection2
      Counter-Loop2
            {//body of Counter-Loop2 }
}
Option2-Selection2 {
      //body of Option2-Selection2
}
```

Fig. 4: Example of code structure for program P2

Table 6: Generated test cases based on an example of Case 5 for Selection → Counter-Loop

Test Case	Input Conditions
TC1	Valid-Option1-Selection1, Valid-Loop1
TC2	Valid-Option1-Selection1, Invalid-Loop1
TC3	Valid-Option1-Selection1, Illegal-Loop1
TC4	Invalid-Option-Selection1
TC5	Illegal-Option-Selection1
TC6	Valid-Option1-Selection2, Valid-Loop2
TC7	Valid-Option1-Selection1, Invalid-Loop2
TC8	Valid-Option1-Selection1, Illegal-Loop2
TC9	Valid-Option2-Selection2
TC10	Invalid-Option-Selection2
TC11	Illegal- Option-Selection2

2.3.3 SentinelLoop and Selection (consecutive)

The way of deriving test cases for this design structure is exactly the same as CounterLoop and Selection (consecutive).

2.3.4 SentinelLoop and Selection (nested)

This type of design structure, the means of deriving test cases is the same as CounterLoop and Selection (nested). However, in terms of the numbers of test data involved in each test case for all design structures, they will be different. Sub-section (B) has described about the numbers of test data for both types of loops either they are arranged consecutively or in a nested way.

3 An Example of Derived Test Set

In order to understand how the design of test set maps to APA, the following example illustrates the concept. Fig. 5 depicts the sample of programming exercise and its functional specifications, and Fig. 6 is its respective flow-graph representation. Table 7 shows the derived test set, which includes test cases that cover valid, invalid and illegal path conditions.

Question:
Write a program that reads an age of a person, which is an integer, and print the status of the age that is based on the following:

age	status
$0 \leq age \geq 21$	"Youth is a wonderful thing. Enjoy"
age > 21	"Age is a state of mind. Enjoy"

Functional specification:
Input – an age, which is an integer value
Output – status of the age, which is a String
Functional Process:
-If the age is an integer and it fulfils one of the listed condition ($0 \leq age \geq 21$ or age > 21), the program shall return the corresponding status of the age as the program output.
-If the age value is less than zero or a character, the program return a null value.
-If the age value is a String, the program return an exception error message.

Fig. 5: Sample of programming exercise and its functional specifications

Based on Fig. 6, the program produces two linearly independent paths which are; *Path1* → b, a1, a2, e and *Path2* →b, a1, a3, e. For this study, these paths cover the valid path conditions and they are compulsory to be exercised because they are commonly a part of program specifications in APA. The same applies to the path that does not fulfil either *Path1* or *Path2* that is the path of b, a1, e, which cover invalid path conditions. Both valid and invalid path conditions fall into positive testing criterion.

Based on Table 7, the test cases of TC1, TC2 and TC3 represent positive testing criteria (or test data adequacy-reliability) and the test case of TC4 covers

negative testing criteria (or test data adequacy-validity). Specifically, TC1 and TC2 cover the valid path conditions and TC3 cover the invalid path condition. As the parameter of *age* is an integer data type, a String value is used to cover the illegal path condition. This condition results the program under testing returns an exception error, which determines it is the point where an error occurs due to the input-mismatch exception. Considering the flow graph in Fig. 6, the illegal path condition can take part as long as the test datum to represent the parameter of *age* is a non-integer data type. Although a number of test cases can cover such path conditions, this study merely selects one value of test datum to represent a single type of exception error. It is mainly to reduce the overall number of test cases generated especially in the case of the number of input variable/parameter increased significantly. It is adequate in APA since it covers the negative testing criteria.

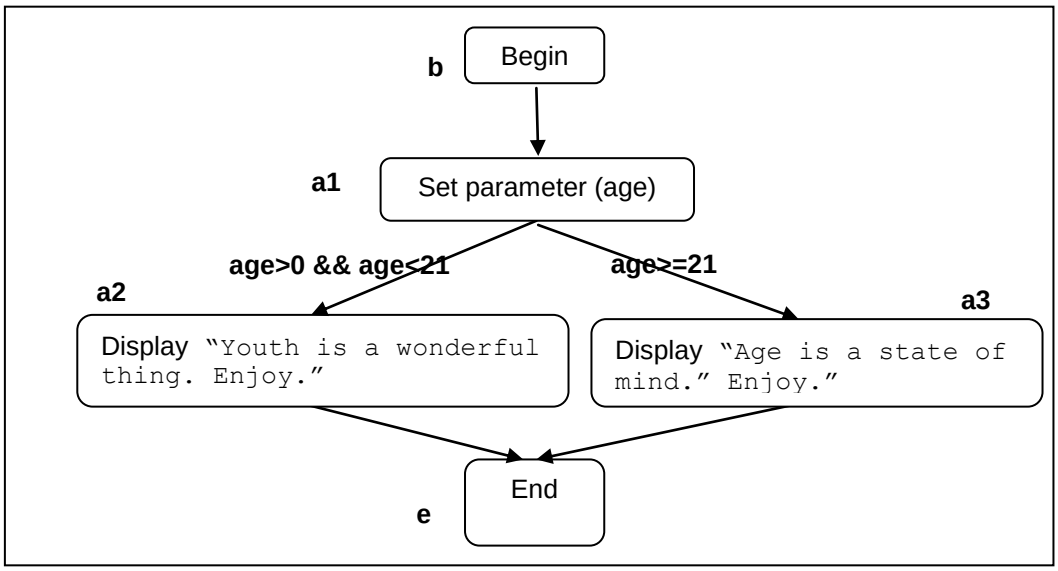

Fig.6: Flow graph that represents the fragment of code shown in Fig. 5

Table 7: Schema of test set for the fragment of code in Fig. 6

Test Case (TC)	Input	Test Case Description	Path Covered	Path Condition
	age			
TC 1	18	Exercise the branch of age>0 && age<21	b, a1, a2, e	Valid branch of age>0 && age<21
TC 2	45	Exercise the branch of age>21	b, a1, a3, e	Valid branch of age>21
TC 3	-4	null	b, a1, e	Invalid branches of age>0 && age<21and invalid age>21
TC 4	"abc"	Error and the program terminate	None	Illegal path

4 Results and Discussion

In order to measure the completeness coverage of the test data adequacy of FaSTDG approach (focuses the structural testing) in terms of the criteria of reliability and validity test data adequacy, we conducted a controlled experiment that employs the one-group *pretest-posttest* design. It is an experimental design in which a single group is measured or observed before and after being exposed to a treatment [48]. Fig. 7 depicts the design of the experiment.

O (*Pre-test*)	*X* (Treatment)	*O* (*Post-test*)
The criteria of reliability and validity to measure the completeness coverage of test data adequacy of structural test data generation in programming assessment (Current Method)	FaSTDG Approach (FaSt-generator)	The criteria of reliability and validity to measure the completeness coverage of test data adequacy of structural test data in programming assessment

Fig. 7: Design of the Controlled Experiment (adapted from [48])

Based on Fig. 7, the symbol *X* represents exposure of the group to the treatment of interest (independent variable), while *O* refers to the measurement of dependent variable. The *pre-test* experiment intends to measure the degree of the completeness coverage of the criteria of test data adequacy (reliability and validity) for *Current Method* in preparing a set of test data to perform the dynamic structural testing in programming assessment. The *Current Method* refers to the means of preparing test data based on the individual user's knowledge in a certain test case design**.**

This experiment used three samples of programming exercises as assignments in the scenario setting that each subject should follow. The exercises cover the three main control structures in Java programming, which are sequential, selection and repetition (loop). They are the most important concepts of programming that every student of elementary of programming course should master. In addition, they are commonly included as the main objective to be achieved in the course syllabus. The subjects of the controlled experiment were lecturers who have been teaching the course of Introduction to Programming (STIA1013) at UUM. There were all, a total of twelve (12) subjects. They at least have been teaching the programming course for one semester.

We used a set of *pre-test* and *post-test* questions that consisted of the same content. Each set of the question consisted of two sections: Section A: Testing of dynamic correctness – reliability and validity of test data adequacy and Section B: Testing of dynamic correctness – test case coverage. It included a combination of close-ended and open-ended items or questions. All items in Section A are close-ended. Whereas, all items in Section B are open-ended. We included both the criteria of positive and negative testing [49] as items in Section A. The criterion of

positive testing (or test data adequacy-reliable) involves two statements. They are "*the program does what it is supposed to*" and "*the program does what it is not supposed to do*". However, negative testing criterion (or test data adequacy-valid) concerns the statement of "*the program does not do anything that is not supposed to do*". Section A used three scales; *C* (provides total evaluation of metric), *P* (provides partial evaluation of metric), and *I* (provides inconclusive evaluation of metric). This study adopts the scales from Boehm *et al.* [50] as Section A attempts to cover the completeness of the two criteria as items in the question.

Section B consisted of open-ended items, which are related to the derived test cases for the three samples of programming exercises used in the experiment. For each of the exercises, the test cases should be recorded in a listing format consisting of their input values (test data), the outputs produced and brief descriptions of each test case. The reason of using open-ended items is that every lecturer may have different levels of knowledge in the test case design to derive a set of test data used to assess students' programming solutions in terms of the structural testing.

The following sub-sections discuss the results of the experiment:

4.1 Testing of dynamic correctness – reliability and validity test data adequacy

As stated in the *pre-test* question, the evaluation of metric refers to the coverage of the criteria of test data adequacy employed either it is *complete*, *partial* or *inconclusive* evaluation of metric. In this experiment, the term "*partial*" means that in deriving test data, the lecturers cover more than 10% of the criteria covered by FaSTDG approach. While "*inconclusive*" denotes that the lecturers cover the criteria of FaSTDG approach less than 10%. The term "total" refers to covering the same criteria as FaSTDG approach.

Both Fig. 8 and Fig. 9 tabulated the results in terms of the test data adequacy covered in the practice of programming assessment for structural testing. Fig. 8 emphasised the positive testing criteria (the criteria of reliable test data adequacy). It shows that for both the criteria of "*the program does what it is supposed to do*" and "*the program does what it is not supposed to do*", about 58% and 75% of the subjects respectively provided the partial evaluation of the metric. These outdo the results in terms of providing the metric totally or inconclusively. In case of structural testing, they were about 42% of the subjects who totally covered the evaluation metric. Only 17% of the respondents provided inconclusive evaluation of the metric for the criteria of "*the program does what it is not supposed to do*". In terms of the negative testing criterion (the criteria of "*the program does not do anything that is not supposed to do*"), it seemed to show the similar trend as the positive testing criterion (see Fig. 8). The highest rating is still the category of providing the metric partially, in which it has a total of 67%. Similar to the criteria of "*the program does what it is supposed to do*" and "*the program does what it is not supposed to do*", the criteria of "*the program does not do anything that is not*

supposed to do" had one subject who totally provided the evaluation metric in deriving test data. Thus, in overall it can be concluded that FaSTDG approach provides an adequate set of test data to cover the structural testing of programming assessments.

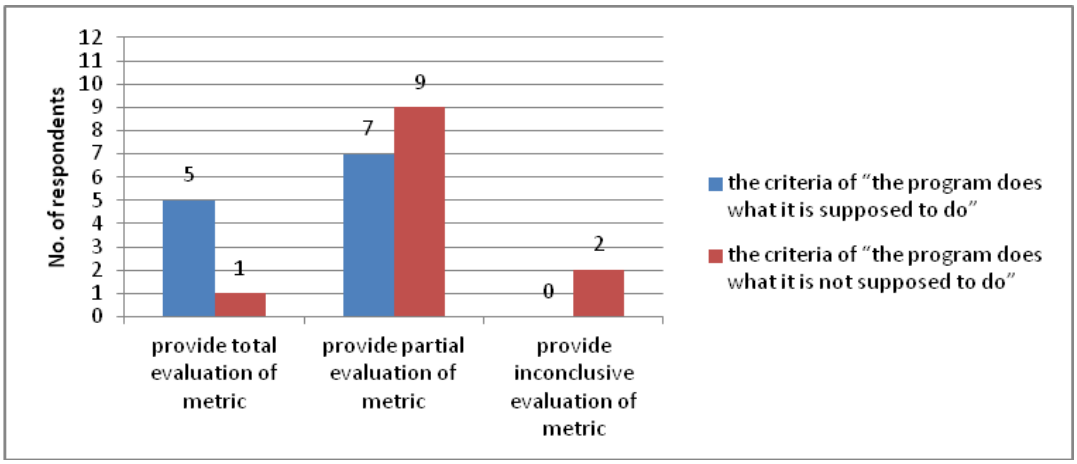

Fig. 8: Number of subjects who derived test data for structural testing based on positive testing criterion

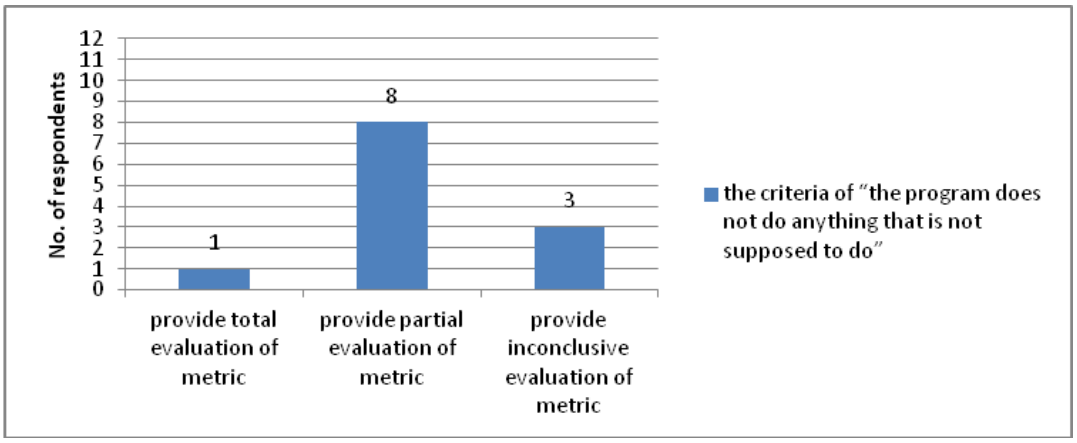

Fig. 9: Number of subjects who derived test data for structural testing based on negative testing criterion

4.2 Testing of dynamic correctness – test case coverage

This sub-section reveals in detail the result in terms of the coverage of test cases for each of the programming exercises used as samples in the conducted experiment. The results were reported in line charts that were based on the frequency of test cases covered for the respective samples of programming exercises and the criteria of test data adequacy (reliable or valid) considered. Fig.10 illustrates the results to compare the coverage of test cases derived from the current methods used in the practice of programming assessment, with the one

that were derived from FaSTDG approach. The data of this experiment were recorded based on the respective test data adequacy covered in deriving test data.

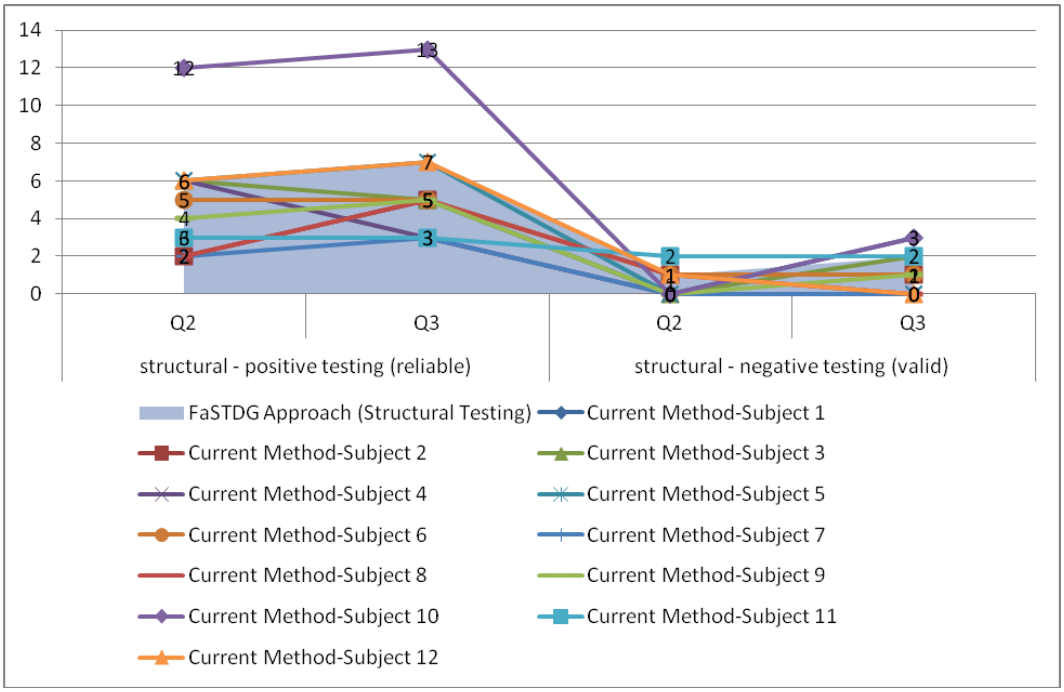

Fig. 10: The results to compare the coverage of test cases derived from the current methods used in the practice of programming assessment, with the one that were derived from FaSTDG approach

From the figure, the results show that structural testing with the criteria of positive testing (or test data adequacy-reliable), eleven (11) of the subjects derived the test data similar or less than one to four in the number of test cases derived as compared to FaSTDG approach. However, only one subject appears to have the derived test data more than that of by FaSTDG approach (differ from 6 to 7 in the number of test cases derived). As question Q2 involved a selection control structure with five options, the subject seemed to consider the values of $-\alpha$ (just less) and $+\alpha$ (just greater) at two boundaries of the range values. It is similar to the concept of original boundary value analysis (BVA). Such conditions may increase the overall number of test cases generated. In this study, these test cases have been included to cover functional testing. In addition, almost all the test cases that cover negative testing criterion returned the same type of exception error. Therefore, this study employs the path coverage criteria so that it is adequate to cover the branches that their input conditions are valid (true) and invalid (false) to suit the APA context.

In terms of negative testing criterion or test data adequacy-valid, all the results are just the same or differ one (greater or less) in the number of test cases derived by FaSTDG approach. Thus, these results do not show a significant difference in

term of the current methods used in the practice of programming assessment as compared to FaSTDG approach. In overall, it concludes that FaSTDG approach provides an adequate set of test data (considering the criteria of reliable and valid test data adequacy) to perform the structural testing of a program for APA. Question Q1 did not involve in structural testing as the exercise just covers a sequential control structure (particularly for functional testing).

In conclusion, it is proved that FaSTDG approach derives and generates an adequate set of test data to be used to perform the structural testing of a program for APA.

5 Related Work

Structural testing is the most common form of assessment to determine the coverage of the program logic and it must be executed as least once such as statement coverage, path coverage, branch (or decision) coverage, condition coverage and decision/condition coverage [51]. For this technique, the test data are driven by examining the logic (or implementation) of a program, without concerning its requirements [52]. Structural testing becomes a better choice to provide more thorough testing in programming assessment to compliment functional testing. The same as functional testing, this testing also falls into dynamic testing category.

The test data to perform correctness assessment of structural testing can be automatically generated or manually provided. The number of studies that focuses this aspect of assessment is relatively small compared to functional testing. Among the related studies that provided fixed test data manually are [53][54][55][36][11][37][56]. Some of these studies utilize JUnit to guide the testing. Thus far, it appeared that only a study by Cheng *et al.*[35] uses an external tool that utilizes JUnit to generate the desired test data automatically.

To date, in term of structural testing, limited studies have attempted to automate the process of generating test data that exclude the use of particular lecturers' knowledge in test cases design. A study by Ihantola [32] was among the earliest study to bring formally justified test data generation and education closer to each other. The study utilizes symbolic execution in Java PathFinder (JPF) in deriving the test data. The results of the study were also reasonable and well applied in other contexts than automatic assessment of programming exercises. While, a study by Tilmann *et al.* [33] has yet appeared to be the latest work. This study focuses more on an interactive-gaming-based teaching and learning for introductory to advanced programming or software engineering courses. However, integrating test data generation and APA becomes a part of the proposed work. Thus, it deduces that most of the existing studies generate test data manually. However, the efforts to prepare test data automatically is still questionable in some ways, particularly with regard to conforming to a certain level of errors point coverage (or negative testing criteria).

6 Conclusion and Future Work

This paper has presented the means of designing test set guided by the improved path coverage criterion particularly to perform structural testing of a program in APA. In order to furnish an adequate set of test data to conform to specifications of a solution model as well as to include certain extend of error-prone points coverage, we embed positive and negative testing criteria into the test set.

The design of test set provides a guideline to lecturers' of programming courses to generate test set with appropriate test data to perform structural testing. It also includes the necessary criteria employed in practise. Thus, the lecturers do not need to have any specific expertise in the knowledge of test case design. The example included in this study shows that the derived test set and test data do fulfill the criteria of an ideal test criterion that is both reliable and valid [57]. Also, based on the results collected from the conducted experiment as discussed earlier, it can be deduced that the criteria of positive and negative testing (test data adequacy- reliable and valid) are adequate as they cover what have been applied is the practice of programming assessments.

In recent years, it has appeared that applying meta-heuristic search techniques particularly for automatic test data generation was of burgeoning interest to many researchers. On particular interest, McMinn [58] analyzed the details of the results of the survey in such techniques. Also, in our previous work [59], we reported the statistics and trends of the studies in automated test data generation within the year of 1976 and 2010. The result shows that meta-heuristic algorithms have become the popular approaches applied since early 2000 as there have been increasing demands on finding the most optimum test data so as to perform software testing efficiently (cost reduction). From the survey, it also depicts that genetic algorithm was among the accepted meta-heuristic search techniques-applied. In addition, the result also reveals that application of meta-heuristic search techniques seems to be the most popular adapted technique for structural test data generation. Meta-heuristic techniques are the high-level frameworks that utilize heuristics in order to find solutions to combinatorial problems at a reasonable computational cost [58]. Among the most popular meta-heuristic techniques that have been employed in test data generations are gradient descent, Simulated Annealing (SA), Ant Colony Optimization (ACO), Tabu Search and evolutionary algorithms such as Genetic Algorithm (GA).

As the future work, in order to realize the approach so that it can be generalized and to be applicable in the context of software testing research area (particularly for a large scale of testing), an application of any meta-heuristic algorithm or a hybrid among of them possibly becomes a better solution. The main reason definitely due to the most optimum set of test data is one of the way to ensure the testing process can be undertaken efficiently as one of the main issues and challenges in the area of automated test data generation is the exposure to the NP-hard or non-deterministic polynomial hard problem (the time

complexity of $O(n^n)$). Other latest meta-heuristic algorithms such as Harmony Search, and/or Fire-fly could become a promising alternative as well.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the MOSTI eScience Fund (R.J130000.7928.4S065) of UTM for partly supporting this research.

References

[1] D. Jackson. 1996. A Software System for Grading Student Computer Programs, *Computers and Education*, Vol. 27, No. 3-4, (1996), pp. 171-180.

[2] J. L. F. Aleman. 2011. Automated Assessment in Programming Tools Course, *IEEE Transactions on Education*, Vol. 54, No. 4, (2011), pp. 576-581.

[3] P. Ihantola, T. Ahoniemi, and V. Karavirta. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments, In Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10), pp. 86-93.

[4] S. C. Shaffer. 2005. Ludwig: An Online Programming Tutoring and Assessment System, *ACM SIGCSE Bulletin*, Vol. 37, No. 2, (2005), pp. 56-60.

[5] G. Tremblay and E. Labonte. 2003. Semi-Automatic Marking of Java Programs using JUnit", In Proceeding of International Conference on Education and Information Systems: Technologies and Applications (EISTA '03), pp. 42-47.

[6] D. Jackson and M. Usher. 1997. Grading Student Programs using ASSYST, In Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, pp. 335–339.

[7] M. Luck and M. S. Joy. 1999. Secure On-line Submission System, *Journal of Software – Practise and Experience*, Vol. 29, No. 8, pp. 721-740.

[8] M. Blumenstein, S. Green, A. Nguyen and V. Muthukkumarasamy. 2004. GAME: A Generic Automated Marking Environment for Programming Assessment, In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04), Vol. 2, pp. 212-216.

[9] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppala and P. Silvasti. 2004. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2, *Informatics in Education*, Vol, 3, No. 2, pp. 267-288.

[10] M. Choy, U. Nazir, C. K. Poon and Y. T. Yu. 2005. Experiences in Using an Automated System for Improving Students' of Computer Programming, Lecture Notes in Computer Science Learning. Springer Berlin/ Heidelberg, pp. 267 – 272.

[11] N. Truong, P. Bancroft and P. Roe. 2005. Learning to Program Through the Web, *ACM SIGCSE Bulletin*, Vol. 37, No. 3, pp. 9-13.

[12] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. 2006. Automated

Assessment and Experiences of Teaching Programming, *Journal of Educational Resources in Computing*, Vol. 5, No. 3, Article 5.

[13] J. Baldwin, E. Crupi and T. Estrellado. 2006. WeBWork for Programming Fundamentals, In Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Bologna, Italy, pp. 361-361.

[14] B. Auffarth, M. Lopez-Sanchez, J. C. Miralles, A. and Puig. 2008. System for Automated Assistance in Correction of Programming Exercises (SAC), In Proceedings of the fifth CIDUI - V International Congress of University Teaching and Innovation.

[15] G. Tremblay and E. Labonte. 2003. Semi-Automatic Marking of Java Programs using JUnit, In Proceeding of International Conference on Education and Information Systems: Technologies and Applications (EISTA '03), Orlando, Florida, pp. 42-47

[16] A. Nunome, H. Hirata, M. Fukuzawa and K. Shibayama. 2010. Development of an E-learning Back-end System for Code Assessment in Elementary Programming Practice, In Proceeding of the 38th Annual Fall Conference on SIGUCCS, Norfolk, VA, USA, pp. 181-186.

[17] R. Queiros and J. S. Leal. 2012. PETCHA- A Programming Exercises Teaching Assistant, In Proceeding of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12), Haifa, Israel, pp. 192-197.

[18] F. A. Shamsi and A. Elnagar. 2012. An Intelligent Assessment Tool for Students' Java Submissions in Introductory Programming Courses, *Journal of Intelligent Learning Systems and Applications*, Vol. 4, No. 1, pp. 59-69.

[19] M. Sherman, S. Bassil, D. Lipman, N. Tuck and F. Martin. 2013. Impact of Auto-Grading on an Introductory Computing Course, *Journal of Computing Sciences in Colleges*, Vol. 28, No. 6, pp. 69-75.

[20] L. Malmi, R. Saikkonen and A. Korhonen. 2002. Experiences in Automatic Assessment on Mass Courses and Issues for Designing Virtual Courses, In Proceedings of The 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE' 02), Aarhus Denmark, pp. 55-59.

[21] H. D. Chu, J. E. Dobson and I. C. Liu. C. 1997. FAST: A Framework for Automating Statistical-based Testing, *Software Quality Journal*, Vol. 6, No. 1, pp. 13-36.

[22] L. A. Clarke. 1976. A System To Generate Test Data and Symbolically Execute Programs, *IEEE Transaction on Software Engineering.*, SE-2(3), pp. 215-222.

[23] N. Gupta, A. P. Mathur and M. L. Soffa. 1998. Automated Test Data Generation Using an Iterative Relaxation Method, *ACM SIGSOFT Software Engineering Notes*, Vol. 23, No. 6, pp. 231-245.

[24] R. P. Pargas, M. J. Harrold and R. R. Peck. 1999. Test-Data Generation Using Genetic Algorithms, *Journal of Software Testing, Verification and Reliability*, Vol. 9, No. 4, pp. 263-282.

[25] J. Offutt, S. Liu, A. Abdurazik and P. Ammann. 2003. Generating Test Data from State-Based Specifications, *Software Testing, Verification And Reliability*, Vol. 13, pp. 25–53.

[26] K. Z. Zamli, N. A. M. Isa, M. F. J. Klaib and S. N. Azizan. 2007. Tool for Automated Test Data Generation (and Execution) Based on Combinatorial Approach, *International Journal of Software Engineering and Its Applications*, Vol. 1, No. 1, pp. 19-36.

[27] M. Alshraideh, L. Bottaci and B. A. Mahafzah. 2010. Using program data-state scarcity to guide Automatic Test Data Generation, *Software Quality Journal*, Vol. 18, No.1, pp. 109-144.

[28] Y. Zhang, D, Gong and Y. Luo. 2011. Evolutionary Generation of Test Data for Path Coverage with Faults Detection, In Proceeding of the 2011 Seventh International Conference on Natural Computation (ICNC), Vol. 4, pp. 2086-2090.

[29] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun and J. Wegener. 2012. Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation, *IEEE Transactions on Software Engineering*, Vol. 38, No. 2, pp. 453-477.

[30] T. Benouhiba and W. Zidoune. 2012. Targeted adequacy criteria for search-based test data generation, In Proceeding of 2012 International Conference on Information Technology and e-Services (ICITeS'12), Sousse, Tunisia, pp. 1-6.

[31] Z. Shukur, Z., R. Romli and A. B. Hamdan. 2005. Skema Penjanaan Data dan Pemberat Ujian Berasaskan Kaedah Analisis Nilai Sempadan (A Schema of Generating Test Data and Test Weight Based on Boundary Value Analysis Technique), *Technology Journal*, Issue 42(D), June 2005, pp. 23-40.

[32] P. Ihantola. 2006. Automatic Test Data Generation for Programming Exercises with Symbolic Execution and Java PathFinder, Master Thesis of Helsinki University of Technology, Finland.

[33] N. Tillmann, J. D. Halleux, T. Xie, S. Gulwani and J. Bishop. 2013. Teaching and Learning Programming and Software Engineering via Interactive Gaming, In Proceedings of the 2013 International Conference on Software Engineering (ICSE'13), San Francisco,CA, USA, pp. 1117-1126.

[34] M. Guo, T. Chai and K. Qian. 2010. Design of Online Runtime and Testing Environment for Instant Java Programming Assessment, In Proceeding of 7th International Conference on Information Technology: New Generation (ITNG 2010), Las Vegas, NV, pp. 1102-1106.

[35] Z. Cheng, R. Monahan and A. Mooney. 2011. nExaminer: A Semi-automated Computer Programming Assignment Assessment Framework for Moodle, In Proceedings of International Conference on Engaging Pedagogy 2011 (ICEP11) NCI, Dublin, Ireland, pp. 1-12.

[36] S. H. Edwards. (2003). Improving Student Performance by Evaluating How Well Student Test Their Own Programs, *Journal on Educational Resources in Computing (JERIC)*, Vol. 3, No. 3, pp. 1-24.

[37] G. Fischer and J. W. Gudenberg. 2006 Improving the Quality of Programming Education by Online Assessment, Proceedings of the 4th International Symposium on Principles and Practice of programming in Java, Mannheim, Germany, pp. 208-211.

[38] O. Gotel, C. Scharff and A. Wildenberg. 2007. Extending and Contributing to an Open Source Web-Based System for the Assessment of Programming Problems, In Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ'07), Lisboa, Portugal, pp. 3-12.

[39] F. Jurado, M. Redondo and M. Ortega. 2012. Using Fuzzy Logic Applied to Software Metrics and Test Cases to Assess Programming Assignments and Give Advice, *Journal of Network and Computer Applications*, Vol. 35, No. 2, pp. 695-712.

[40] E. L. Jones. 2001. Grading Student Programs- A Software Testing Approach, *Journal of Computing Sciences in Colleges*, Vol. 16, No. 2, pp. 185-192.

[41] J. Isong. 2001. Developing An Automated Program Checker, *Journal of Computing Sciences in Colleges*, Vol. 16, No. 3, pp. 218-224.

[42] K. A. Foster. 1980. Error Sensitive Test Cases Analysis (ESTCA), *IEEE Transactions on Software Engineering*, SE-6 (3), pp. 258-1980.

[43] R. Romli, S. Sulaiman and K. Z. Zamli. 2011. Test Data Generation in Automatic Programming Assessment: The Design of Test Set Schema for Functional Testing, In Proceeding of 2nd International Conference on Advancements in Computing Technology (ICACT'11), Jeju Island, South Korea, pp. 1078-1082.

[44] D. S. Malik and R. P. Burton. 2009. Java Programming: Guided Learning with Early Objects, Course Technology, Booston USA.

[45] J. Lewis, P. DePasquate and J. Chase. 2008. Java Foundations: Introduction to Program Design and Data Structures, Pearson Education, Inc, USA.

[46] D. Y. Liang. 2009. Introduction to JAVA Programming, 7th Edition, Pearson Education Inc., New Jersey.

[47] J. W. Howden. 1975. Methodology for Generation of Program Test Data, *IEEE Transactions on Computers*, C-24 (5), pp. 554-560.

[48] J. R. Fraenkel and N. E. Wallen. 2000. How to Design and Evaluate Research in Education, 4th Edition, McGraw-Hill Companies, Inc, U.S.A.

[49] R. Romli, S. Sulaiman and K. Z. Zamli. 2011. Current Practices of Programming Assessment at Higher Learning Institutions, Part 1, CCIS 179, Springer Berlin/Heidelberg, pp. 471-485.

[50] B. W. Boehm, J. R. Brown and M. Lipow. 1976. Quantitative Evaluation of Software Quality, In Proceedings of the 2nd Iinternational Conference on Software Engineering (ICSE '76), U.S.A., pp. 592–605.

[51] R. A. DeMillo and A. J. Offutt. 1991. Constraint-Based Automatic Test Data Generation, *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pp. 900-910.

[52] J. Lewis, P. DePasquate and J. Chase. 2008, Java Foundations: Introduction to Program Design and Data Structures, Pearson Education, Inc, USA.

[50] D. Jackson. 1996. A Software System for Grading Student Computer Programs, *Computers and Education*, 27 (3-4), pp. 171-180.

[53] Z. Shukur. 1999. The Automatic Assessment of Z Specification, PhD Thesis. University of Nottingham, UK.

[54] C. A. Higgins, P. Symeonidis and A. Tsintsifas. 2002. The Marking System for CourseMaster, In Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '02), Aarhus, Denmark, pp. 46-50.

[55] D. J. Malan. 2013. CS50 Sandbox: Secure Execution of Untrusted Code, In Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '13). Denver, CO, pp. 141-146.

[56] N. Tillmann and J. D. Halleux. 2008. Pex-white Box Test Generation for .NET, Tests and Proofs, Lecture Notes in Computer Science, Vol. 4966, pp. 134-153.

[57] J. B. Goodenough and S. L. Gerhart. 1975. Towards a Theory of Test Data Selection, In Proceedings of the International Conference on Reliable Software, New York, USA, pp. 493-510.

[58] P. McMinn. 2004. Search-based Software Test Data Generation: A Survey, *Software Testing, Verification & Reliability*, Vol. 14, No. 2, pp. 105-156.

[59] R. Romli, S. Sulaiman and K. Z. Zamli. 2010. Automatic Programming Assessment and Test Data Generation: A Review on Its Approaches, In Proceeding of 2010 International Symposium on Information Technology (ITSim'10), Kuala Lumpur, M'sia, 2010, pp. 1186-1192.