 Copyright © Long H. Pham, Nam P. Mai, Mai H. Dinh, Tho T. Quan, Hung Q. Ngo, 2013

Assisting Students in Finding their Own Bugs in Programming Exercises using Verification and Group Testing Techniques

Long H. PHAM
University of Technology, Ho Chi Minh City, Vietnam
longph@cse.hcmut.edu.vn

Nam P. MAI
University of Technology, Ho Chi Minh City, Vietnam
bkiter09@gmail.com

Mai H. DINH
University of Technology, Ho Chi Minh City, Vietnam
maidh91@gmail.com

Tho T. QUAN
University of Technology, Ho Chi Minh City, Vietnam
qttho@cse.hcmut.edu.vn

Hung Q. NGO
State University of New York at Buffalo, New York, USA
hungngo@buffalo.edu

ABSTRACT

We combine theorem proving with group testing to develop an online intelligent tutoring system that can automatically verify students' programming exercises without running their programs. In particular, our system can indicate suspicious portions in programs which may cause logical errors. This system is basically platform-independent, which can be adapted to teaching any imperative language like C, C++ or Java. Group testing, on the other hand, helps us reasonably locate the programs' potential portions that are logically wrong. The experiments show that our system can detect error parts in programs quite well. So our system can act like a virtual tutor in programming courses. It should be very useful for any distant learning programs, which these days are widely in use.

Keywords: *Intelligent Tutoring System, Programming Exercises, Program Verification, Bugs Locations, Group Testing.*

I. INTRODUCTION

Programming courses are essential for any computer science study. To master programming skills, experience shows that practising with problems is the best method. In traditional education, the best practice applied so far is that class tutors have to read students' programs to verify their correctness. However, there are often too many programs that need to be verified, and reading others' code is very error-sensitive. Thus, traditional education becomes less effective in programming courses.

An automated assessment system is a good solution for this problem. Most of these systems check whether the students' programs can pass all test cases in an automatically generated test suite or not (Douce et al., 2005; Ala-Mutka, 2005; Ihantola et al., 2010; Kaushal & Singh, 2012; Jurado et al., 2012). However, this approach has some disadvantages. Firstly, the test suite must be large enough to cover all possible errors in the program. In addition, executing a possibly bugged program is potentially dangerous for the system.

To overcome such obstacles, *static methods* are proposed to verify programs' correctness statically without running programs. In Quan et al. (2009), two static methods of *theorem proving* and *model checking* are combined to build a web-based tutoring system. These methods are also called *formal methods*, which means that they use mathematics-based techniques to check the program's properties. While theorem proving can verify the program's correctness, model checking can generate counter examples to help trace down the bugs via the corresponding execution flows if the program is false. The analysed results are then shown to students.

However, this system can only help learners to become aware of a program's correctness. It cannot help to effectively locate the *root causes* of the problems since the generated counter examples are too complicated for students to follow. Moreover, determining the root cause locations alongside the execution flows provided by the counter examples is non-trivial, especially for novice programming learners.

Thus, it is intuitively more convenient if we can identify the parts of the program that most likely contain the root causes and show them to students for further investigation. In industry, some *fault localisation* methods can serve this purpose. However, they usually consume too many resources, which makes them unsuitable in an education environment. This motivates us to develop a framework in which we combine theorem proving with *group testing* to achieve the goal. Whereas the theorem proving technique is quite efficient to check the program's correctness, group testing is a simple yet powerful technique which can locate the parts in the program containing bugs. Using group testing is a practical approach since it helps us avoid involving complicated and commercial tools of bug localisation which are not optimum for the learning environment in Vietnam at the moment. The experiments show that our group testing technique can detect error parts correctly in 88% of cases in real non-trivial programming exercises.

The rest of this paper is organised as follows. Some fault localisation methods are discussed in Section II. Then Section III presents our proposed framework. In Section IV we discuss the group testing technique and show how to use it to identify suspicious portions in programs. A case study is given in Section V. We present our experiments in Section VI. Finally, Section VII concludes the paper.

II. FAULT LOCALISATION METHODS

Fault localisation methods aim to detect error parts, or *bugs*, in a program based on testing. There are two approaches in this field: *spectrum-based* and *model-based*. Whereas spectrum-based methods use statistical information from programs' executions to give suspicious ranks to each statement, model-based methods build a model from the executions and use inference rules to determine error parts (Abreu et al., 2008). Although a model-based approach is more accurate than a spectrum-based one, building the model and using inference rules make implementing a real system too complicated. So most of the works in the fault localisation field focus on a spectrum-based approach.

In a spectrum-based approach, the input is execution of the program with a predefined test suite, and the output is a statement ranking from most to least suspicious. Some spectrum-based methods are presented as follows:

- *FOnly* (Zhang et al., 2012) is a method that uses only failed test cases to rank statements. It calculates failure rate $G(c)$ that statement s is executed c times to get pairs $<c,G(c)>$ for each statement. Then it plots these pairs in a diagram and fits the line through them. The statement that has the steeper line is more likely to contain errors.

- In Jones & Harrold (2005) and Abreu et al. (2009), two methods are introduced, which build a function and use information in executions of the program to give suspicious ranks to the program's statements. The information is the number of passed/failed test cases containing statements and in total.

- In Jeffrey et al. (2008), a method called *value replacement* is presented. It alters values that are used in statements in failed executions and checks whether this alteration produces the correct output. Those statements containing values that are more likely to produce the correct output after the alteration are more likely the error ones.

- Renieris & Reiss (2003) presents some other methods such as set-union, set-intersection and nearest neighbour. In these methods, the system finds the initial set of most suspicious statements based on set operations. Then a search technique called *SDG-ranking* is applied to rank other statements.

Although the above methods can detect error parts in programs, they need a lot of calculation time before giving the answer. Thus, they are suitable for an industrial environment but are not preferable in the education domain. In this paper, we consider another mathematical approach, known as group testing, which can localise the error parts based on test results. Group testing is simple to implement yet can produce reliable results, making it a desired choice for educational applications.

III. THE PROPOSED FRAMEWORK

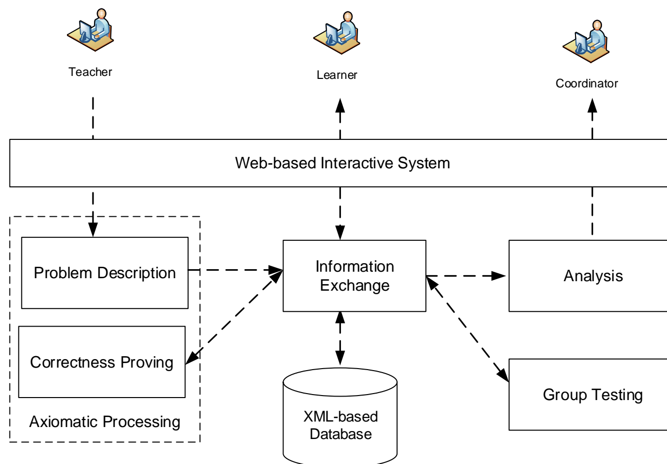

Figure 1: The proposed framework

Figure 1 depicts our proposed framework with three actors: *Teacher*, *Learner* and *Coordinator*. The Teacher's role is providing programming problems. These problems are presented in the Problem Description module. When Learners visit the system, they can try to solve these problems. The Learner's submitted program is verified by the Correctness Proving module. If an error is detected, the program is moved into the Group Testing module to identify error parts. The program's analysed results are given to the Learner. The Coordinator can use the system's information, such as common errors or behaviours of active learners, to assess the course's performance. This information is stored and analysed in the Analysis Module.

Exercise 1	Find the absolute value of a real number. ...
Exercise 2	Find the absolute value of a number (pointer version). ...
Exercise 3	Find the maximum in a pair of 2 real numbers. ...
Exercise 4	Check whether a given integer is odd or even. ...
Exercise 5	Check whether i is divisible by j, given that i and j are 2 integers. ...
Exercise 6	Write a program to convert from METER to INCH. ...
Exercise 7	Write a program to convert from INCH to METER. ...
Exercise 8	Write a program for calculating the diameter of a circle with radius r given as input. ...
Exercise 9	Write a program for calculating the perimeter of a circle with radius r given as input. ...

Figure 2: List of exercises

Problem

Bubble sort, using GroupTesting.

Student code

```
int* sort(int n, int a[])
{
  int i = n - 1;
  while (i > 0)
  {
    int j = 0;
    while (j < i)
    {
      if (a[j] > a[j + 1])
      {
        int temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp + 1;
      }
      j = j + 1;
    }
    i = i - 1;
  }
  return a;
}
```

Submit Reset

Figure 3: A program submitted by student to our system

The Information Exchange module helps other modules exchange data. These data are XML-based and stored as a database for convenience.

The above framework (where Group Testing module is replaced by Model Checking module) has already been implemented as a real web-based system[1]. In this system, there are some predefined programming problems as in Figure 2. Students can choose to implement any problem. Then the website navigates to an interface in which students can write and submit their code, as in Figure 3.

Thus, our framework is considered an improvement of the existing system and is made from two approaches: theorem proving to verify correctness and group testing to identify error parts. Quan et al. (2009) present the details of theorem proving. In the following section, we will discuss the group testing technique.

IV. GROUP TESTING FOR FAULTS LOCATION

A. Group testing

In 1943, Dorfman wanted to test whether any conscripts had syphilis in a very large population of soldiers during WWII (Dorfman, 1943). Instead of individual

[1] http://elearning.cse.hcmut.edu.vn/provegroup/index.jsp

testing, which is very costly in terms of time and effort, the soldiers were divided into groups in a specific way, with each soldier belonging to more than one group. Then blood samples of each group were tested together. If the test outcome for a group was positive, at least one soldier in this group was infected. Otherwise, all members of the group were healthy. More importantly, the test outcome could be used to identify exactly who were the infected individuals. This technique is known as *group testing*.

Suppose we have a group testing strategy with t test samples and N items. We can represent this strategy using a t x N binary matrix, $M = (m_{ij})$ where $m_{ij} = 1$ iff item j belongs to test sample i. We will also use M_i to denote the set of columns corresponding to the 1-entries of row i. Similarly, M^j is used to denote the set of rows corresponding to the 1-entries of column j. In other words, M_i is the ith test sample, and M^j is the set of tests contains item j.

Example 1. *Below is a testing matrix with $t = 4$ and $N = 6$:*

```
1  1  1  0  0  0
1  0  0  1  1  0
0  1  0  1  0  0
0  0  1  0  1  1
```

M_1 is 1st test sample, corresponds to 1st row in testing matrix.

M^1 corresponds to the first column, indicates the set of tests contains 1st item. In that case, only 1st and 2nd tests contain the first item.

Definition 1 (Separable matrix)**.** *A binary matrix M is d-separable if the unions of up to d columns of M are all distinct.*

Definition 2 (Disjunct matrix)**.** *A binary matrix M is d-disjunct if the union of arbitrary $\leq d$ columns does not contain another column.*

Example 2. *In Example 1, all columns in testing matrix are distinct, so it is 1-separable matrix. And it is also 1-disjunct matrix because if we pick up an arbitrary column, this column does not cover any other columns.*

If M is d-separable matrix and the number of positive items in the population is less than or equal to d, we can always exactly determine where they are from the test outcome using non-adaptive combinatorial group testing theory. In particular, if M is d-disjunct matrix, the positive items can be determined effectively using proper decoding algorithms.

Example 3. *Following is a 1-disjunct matrix with 10 items:*

```
1 1 1 1 0 0 0 0 0 0
1 0 0 0 1 1 1 0 0 0
0 1 0 0 1 0 0 1 1 0
0 0 1 0 0 1 0 1 0 1
0 0 0 1 0 0 1 0 1 1
```

Suppose there is only 1 positive item in the population. With this matrix, if the test outcome is {1 1 0 0 0}, the first item is the positive one, since it is the only case that can make the outcomes of the first two test samples positive and the rest negative. Similarly, for any other possible test outcome, we can indicate which item is positive, although there are only 5 test samples used.

B. Fault localisation using group testing

In this section we discuss using group testing to determine fault locations in a program. Firstly, we define a *unit block* of a program, which is a program portion which should not be logically divided into smaller parts when locating bugs. It can be a basic block on a concrete program or an abstracted structure in an abstract program. Let *C* be a program, an ordered set $P_C = \{B_1, \ldots, B_n\}$ where B_i is a unit block of *C* is called an *execution path* of *C* if there exists an input that makes *P* execute from B_i to B_n with the same order as described in P_C.

Example 4. *Suppose we have a function:*

```
int isPositive(int n)
{
  if (n > 0) {
    return X;
  } else {
    return Y;
  }
}
```

Its unit-block representation will be:

```
if S0
  S1
else
  S2
```

Thus, this function has two possible execution paths: (S0, S1) and (S0, S2).

A *testing matrix M* of *C* is defined as a binary matrix $M = (m_{ij})$ where each column M^j corresponds to a unit block B_j of *C*, denoted as B_M^j. A row M_i of

M is considered as a testing path, denoted as P_M^i iff $\exists P_C$ where $\forall j \{B_M^j : m_{ij} = 1\} \subseteq P_C$.

Thus, when we use test cases to test a program P, it can be considered as if we are using a testing matrix M whose rows correspond to the execution paths produced by the test cases when executed on P. In this context, a positive item is a unit block that causes a bug in the program. The process of using matrix M to find bugs on a program P is denoted as $\xi(P, M)$. The complexity of this process depends on the number of unit blocks and the program's structure.

Example 5. *If one can produce two test cases corresponding to $P1$ and $P2$ for the function in Example 2, the testing matrix will be:*

```
S0  S1  S2
1   1   0
1   0   1
```

Suppose the function has at most 1 error (bug). Since the testing matrix is 1-separable, we can determine the error block based on the test outcome. If the test outcome is (0, 0), the function has no bug. If the test outcome is (1, 1), the error block is $S0$. Similarly, if the test outcome is (1, 0) or (0, 1), the error block is $S1$ or $S2$ respectively.

In the white-box testing technique, we try to generate the test cases to cover all of a program's possible execution paths. The strategy we use to generate test cases is based on genetic algorithm (Goldberg, 1989). Firstly, the system generates two test suites randomly. The number of test cases in each test suite is equal to the number of rows in the testing matrix. These test suites will then be crossed over with each other. Test cases corresponding to untested execution paths are kept in the final test suite. Then the system goes into a loop. In each step, a new test suite is generated and crossed over with the current final test suite. Thus, the final test suite will cover the program more and more in each step. This process will terminate when the final test suite covers all the program's execution paths or when the number of repeated steps is over the threshold.

V. CASE STUDY

In this section, we analyse a program in detail. It is an implementation of the bubble sort algorithm.

```
1:  int* sort(int n, int a[])
2:  {
3:    int i = n - 1;                    // block S0
4:    while (i > 0) {                   // block S1
5:      int j = 0;                      // block S2
6:      while (j < i) {                 // block S3
7:        if (a[j] > a[j + 1]) {        // block S4
8:          int temp = a[j];            // block S5
```

```
9:            a[j] = a[j + 1];
10:           a[j + 1] = temp + 1;
11:         }
12:         j = j + 1;                      // block S6
13:       }
14:       i = i - 1;                        // block S7
15:     }
16:     return a;                           // block S8
17: }
```

Listing 1: The case study program

We can see the 10th line is logically wrong. Instead of `a[j + 1] = temp;` it is written as `a[j + 1] = temp + 1;`. Because that line belongs to block S5, we expect S5 should be returned as an error block.

The testing matrix for the above program has 1343 rows and 51 columns. In this matrix, some rows represent paths that do not have corresponding test cases, or the genetic algorithm cannot generate test cases for them, and some columns represent the same blocks because these blocks are repeated in the loop structure. After deleting these rows and compacting each group of columns that represent the same blocks into one column, we have a testing matrix with only 11 rows and 9 columns left.

S0	S1	S2	S3	S4	S5	S6	S7	S8
1	1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1

The test outcome after running test cases is {0 1 0 1 1 1 1 1 1 1 0}. The system compares the test outcome with each column in the testing matrix. Because the test outcome is identical to the S5-column, the system returns S5 as an error block.

The result is displayed in our website as in Figure 4.

With the error block highlighted in red, we believe the programmer can easily see where the problem is and fix it without too much effort.

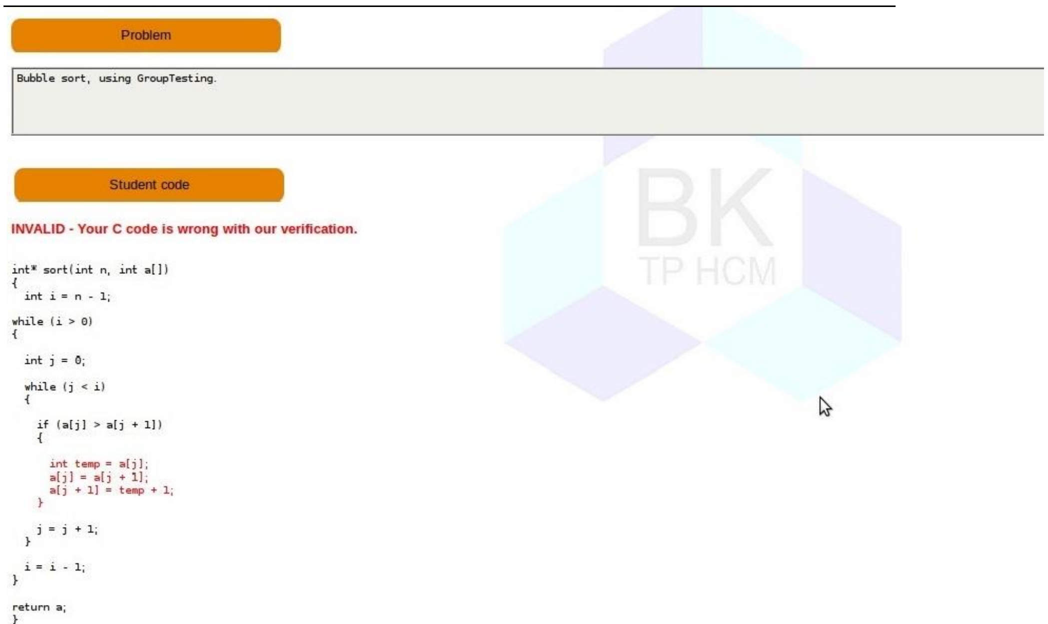

Figure 4: The case study

VI. EXPERIMENTS

We tested our system with six well-known algorithms as presented in Table 1.

In each algorithm, we created some different implementations. Each implementation has exactly one bug. The experiment is used to test whether our system can localise these bugs or not. The system successfully localises the bug in the implementation if the set of returned blocks contains the error block.

Table 1: The experimental algorithms

Number	Algorithms	Descriptions	Number of implementations
I	Finding absolute value	Finding absolute value of a parameter	7
II	Checking odd/even property	Checking whether a parameter is odd or even	3
III	Finding maximum number	Finding the maximum between two parameters	4
IV	Calculating factorial	Finding factorial of a parameter	7
V	Selection sort	Sorting an array using selection sort algorithm	2
VI	Bubble sort	Sorting an array using bubble sort algorithm	2

Example 5. *Below are two implementations of the Finding absolute value Algorithm with bugs.*

Implementation 1:

```
1: int abs(int n) {
2:   if (n >= 0) {
3:     return n;
4:   } else {
5:     return n;                    // should be return -n;
6:   }
7: }
```

Implementation 2:

```
1: int abs(int n) {
2:   if (n >= 0) {
3:     return n + 1;                // should be return n;
4:   } else {
5:     return -n;
6:   }
7: }
```

The result of our experiment is shown in Table 2 and Figure 4. The figure shows the chart comparing right localisations with wrong localisations in each algorithm. The detailed numbers of right/wrong localisations are in the table. As shown in the table, our system can localise error blocks successfully 19 times in a total of 22 implementations. The error blocks in the three remaining implementations are not localised successfully because the generated test suite is not good enough to detect an error in the tested paths, as explained in Example 6.

Example 6. *Below is a wrong implementation of the Finding absolute value Algorithm.*

```
1: int abs(int n) {
2:   if (n >= 5) {
3:     return n;
4:   } else {
5:     return -n;
6:   }
7: }
```

The implementation has two paths and our genetic algorithm can generate test cases to cover both those paths. But to detect an error, the test suite must not only cover all the paths but also contain a test case that is larger than 0 and less than 5. Because the two generated test cases do not belong to this interval, it cannot detect an error in the program, and so the system sees the above program as No error.

This problem of generating best test suite to use in group testing will be addressed in future work based on the constraint-based test-cases generation algorithm in Le et al. (2013), but for now we believe our system can be used to assist students without any serious problems.

Table 2: The experiment results

Algorithms	Number of implementations	Number of right localisations	Number of wrong localisations
Finding absolute value	7	5	2
Checking odd/even property	3	3	0
Finding maximum number	4	3	1
Calculating factorial	7	7	0
Selection sort	2	2	0
Bubble sort	2	2	0
Total	**22**	**19 (88%)**	**3 (12%)**

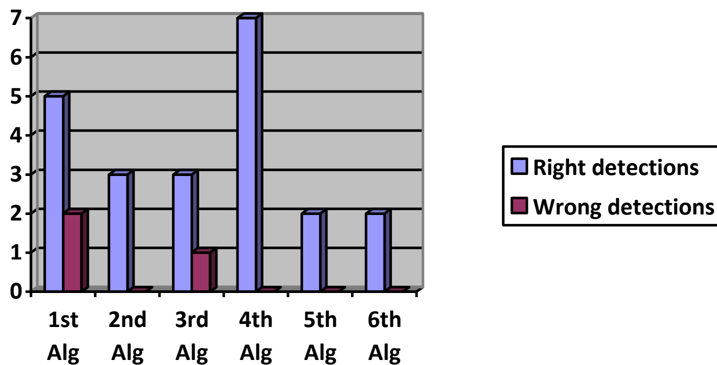

Figure 4: The chart represents the experiment results

CONCLUSION

In this paper we present a framework to verify and identify suspicious portions of programming exercises submitted by students automatically. While verification is done using theorem proving, group testing theory helps identify error blocks. Our framework is tested with 22 versions of 6 algorithms with 88% accuracy. Besides that, our framework can be generalised to any algorithm with a similar structure. In the future, we intend to publish our system to students and use it as a useful tool to help students in programming courses.

REFERENCES

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2008). An observation-based model for fault localization. *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. (pp. 64-70).

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2009). Spectrum-based multiple fault localization. *Proceeding of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, (pp. 88-99).

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education, 15(2)*, 83-102.

Dorfman, R. (1943). The detection of defective members of large populations. *The Annals of Mathematical Statistics, 14*, 436-440.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC), 5*(3), 4.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Chapter 3, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*.. (pp. 86-93).

Jeffrey, D., Gupta, N., & Gupta, R. (2008). Fault localization using value replacement. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. (pp. 167-178).

Jones, J. A., & Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. (pp. 273-282).

Jurado, F., Redondo, M. A., & Ortega, M. (2012). Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. *Journal of Network and Computer Applications, 35*(2), 695-712.

Kaushal, R., & Singh, A. (2012). Automated evaluation of programming assignments. *Proceedings of 2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA)*, (pp. 1-5).

Le, A. D., Quan, T. T., Huynh, N. T., Nguyen, P. H., & Le, N. V. (2013). Combined Constraint-Based Analysis for Efficient Software Regression Detection in Evolving Programs. In M. J. Escalona, J. Cordeiro & B. Shishkov (Ed.), *Software and Data Technologies* (pp. 108-120). Springer Berlin Heidelberg.

Quan, T. T., Nguyen, P. H., Bui, T. H., Huynh, L. V., & Do, A. T. (2009). A framework for automatic verification of programing exercises. *Proceedings of 2nd IEEE International Conference on Computer Science and Information Technology* (pp. 41-45).

Renieris, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on* (pp. 30-39).

Zhang, Z., Chan, W. K., & Tse, T. H. (2012). Fault localization based only on failed runs. *Computer, 45(6)*, 64-71.

Copyright ©2013 IETEC'13, Long H. PHAM, Mai H. DINH, Tho T. QUAN, Hung Q. NGO: The authors assign to IETEC'13 a non-exclusive license to use this document for personal use and in courses of instruction provided that the article is used in full and this copyright statement is reproduced. The authors also grant a non-exclusive license to IETEC'13 to publish this document in full on the World Wide Web (prime sites and mirrors) on CD-ROM and in printed form within the IETEC'13 conference proceedings. Any other usage is prohibited without the express permission of the authors.