Assisting Students in Finding Bugs and Their Locations in Programming Solutions

Long H. PHAM[a], Giang V. TRINH[a], Mai H. DINH[a], Nam P. MAI[a],
Tho T. QUAN[a] and Hung Q. NGO[b]

[a]*Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam*
[b]*State University of New York at Buffalo, New York, United State*

ABSTRACT

Teaching experience shows that programming is time consuming and can be acquired with substantial practice. Besides, students need to know whether their solutions are correct or incorrect and the root causes of their errors. Thus, teaching programming in a large class requires considerably many teaching assistants, which is costly. More importantly, a communication means that can support students virtually anytime is also desirable.
In order to handle these problems, a static method was applied to build an online intelligent tutoring system that can assist students checking their solutions. In addition, when detecting the errors, this system can suggest students to investigate the suspected code. This feature is really significant for students to self-practice and improve their learning.

Keywords: Intelligent Tutoring System, Programming Exercises, Program Verification, Group Testing, Fault Localization

INTRODUCTION

Programming is essential for any computer science study. Research and experience show that practicing with problems is the best method to master the programming skills. In traditional education, class tutors have to read students' programs to verify their correctness. However the number of programs is often too much and code reading is error-sensitive. This explains why the method is less effective in programming courses.

Automated assessment system is a good solution for this problem. This system checks whether the students' programs qualify the test cases in an automatically generated test suite (Ala-Mutka, 2005; Douce et al., 2005; Jurado et al., 2012; Ihantola et al., 2010; Kaushal & Singh, 2012). However, this approach has some disadvantages. For example, the test suite must be large enough to cover all possible errors in the program and executing possibly buggy program is potentially dangerous for the system.

To overcome such obstacles, *static methods* are proposed to statistically verify the programs correctness without running the programs. Two statistical methods referred as *theorem proving* and *model checking* are combined to build a web-based tutoring system (Quan et al., 2009). These methods are also known as *formal methods*, whereby mathematics-based techniques are used to check the program properties. While theorem proving can verify the program correctness, model checking can generate counter examples to help trace down the *faults* (*bugs*) via the corresponding *execution paths* if the program is false.

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

Thus, it is intuitively more convenient if identification of the program parts is identified. This will most likely contain the root causes and can be showed to the students for further investigation. This was the motivation for the researchers to develop a framework in which the above system is combined with *group testing* and *slicing spectrum-based fault localization* to achieve the goal. Group testing is a simple yet powerful method to locate faults in the program, slicing spectrum-based fault localization on the contrary, is more complex. However, it can help to find bugs in case group testing gives the wrong answer. So group testing and slicing spectrum-based fault localization act as complements in the system. The above framework was experimented with non-trivial exercises in Programming Fundamentals course and positive results were recorded. This paper presents the proposed framework. Subsequently, group testing and slicing spectrum-based fault localization are also presented. This is followed by the presentation of two case studies. The final section concludes the paper.

THE PROPOSED FRAMEWORK

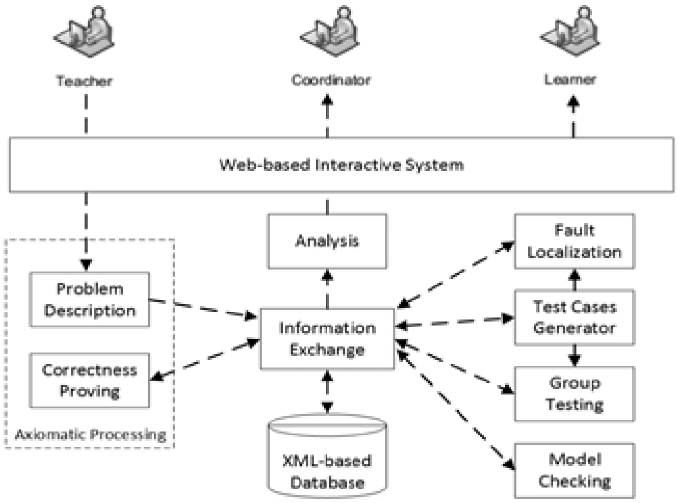
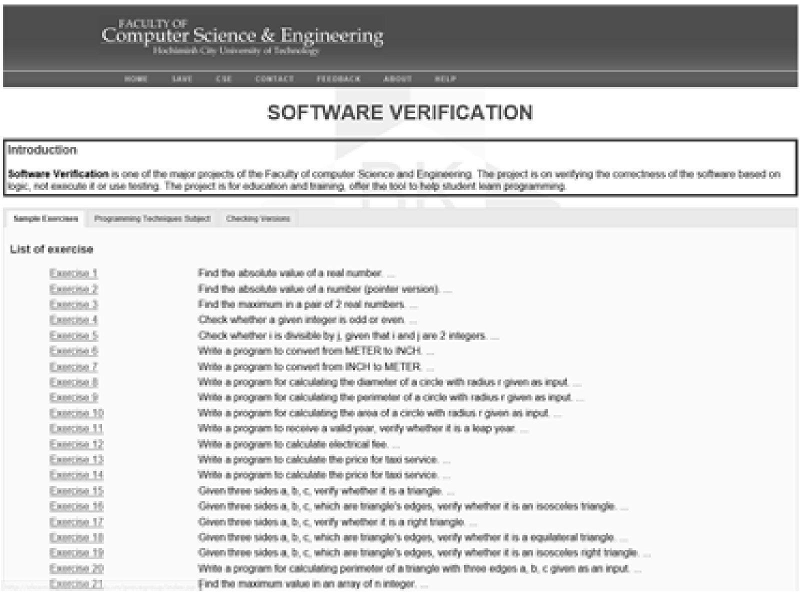


Figure 1. The proposed framework

Figure 1 depicts the proposed framework on three components: Teacher, Learner and Coordinator. The Teacher provides the programming problems presented in the Problem Description Module. When the Learners visit the system, they can try to solve these problems. The Learner's submitted program is verified by the Correctness Proving module. The Model Checking component will identify and show the execution paths which lead to error, if the program is false.

When the Learner wants to find buggy portions in the program, Group Testing and Fault Localization modules will detect them. The process of detecting buggy portions uses test cases generated by Test Cases Generator module. In this module, test cases generated by *constraint-based test-cases generation algorithm* (Le et al., 2013) were combined with random test cases (although test cases are generated randomly, they are chosen to cover different execution paths in the program) to have a suitable test that can identify as many failed execution paths in the program as possible. In the system, instead of actually running the program, a simulator is used to find the program output according to the test cases. With this way, the system is safe and students can be influenced not to use bad programming structures.

The Coordinator can use the system information such as common errors or behaviors of active learners to assess the course performance. This information is stored and analyzed in the Analysis Module. The Information Exchange module helps other modules exchange data. These data are XML-based and they are stored in the database for convenient. The above framework has already been implemented as a real web-based system[i]. In this system, there are some predefined programming problems as listed in Figure 2. The students can choose to implement the problem of their choice. Once selected, the website navigates to an interface in which students can write and submit their code as in Figure 3.


FACULTY OF
Computer Science & Engineering
Hochiminh City University of Technology
HOME
SAVE
CSE
CONTACT
FEEDBACK
ABOUT
HELP
Sample Exercises
Programming Techniques Subject
Checking Versions

SOFTWARE VERIFICATION

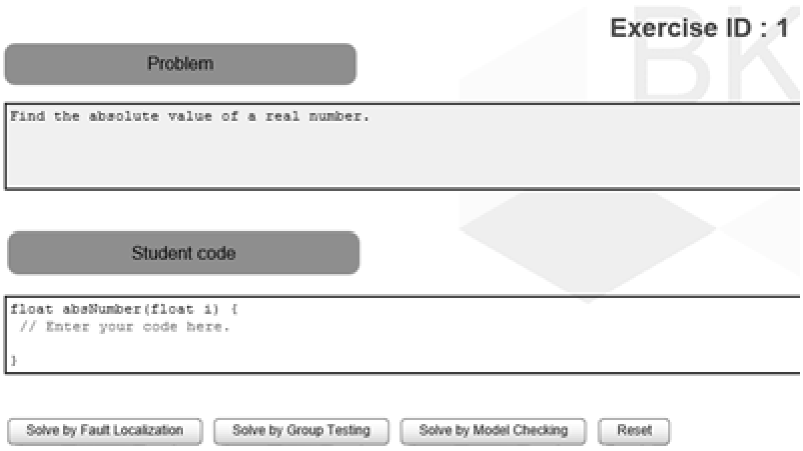
Introduction

Software Verification is one of the major projects of the Faculty of computer Science and Engineering. The project is on verifying the correctness of the software based on logic, not execute it or use testing. The project is for education and training, offer the tool to help student learn programming.

List of exercise

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. ...
Exercise 10	Write a program for calculating the area of a circle with radius r given as input. ...
Exercise 11	Write a program to receive a valid year, verify whether it is a leap year. ...
Exercise 12	Write a program to calculate electrical fee. ...
Exercise 13	Write a program to calculate the price for taxi service. ...
Exercise 14	Write a program to calculate the price for taxi service. ...
Exercise 15	Given three sides a, b, c, verify whether it is a triangle. ...
Exercise 16	Given three sides a, b, c, which are triangle's edges, verify whether it is an isosceles triangle. ...
Exercise 17	Given three sides a, b, c, verify whether it is a right triangle. ...
Exercise 18	Given three sides a, b, c, which are triangle's edges, verify whether it is a equilateral triangle. ...
Exercise 19	Given three sides a, b, c, which are triangle's edges, verify whether it is an isosceles right triangle. ...
Exercise 20	Write a program for calculating perimeter of a triangle with three edges a, b, c given as an input. ...
Exercise 21	Find the maximum value in an array of n integer. ...

Figure 2. List of exercises on Programming Problems


Exercise ID : 1

Problem

```
Find the absolute value of a real number.
```

Student code

```
float absNumber(float i) {
 // Enter your code here.

}
```

Figure 3. The interface for writing and submitting code

The framework is considered as an improvement of the existing system and is made from three main components: theorem proving to verify correctness; model checking to show failed execution paths; group testing and fault localization to identify buggy parts. Quan et al. (2009) present the details of theorem proving and model checking. In the following sections, group testing and slicing spectrum-based fault localization are discussed in more detail.

GROUP TESTING

Basic theory

In 1943, Dorfman wanted to test whether any troop member had contracted the syphilis disease among the large population of soldiers during WWII (Dorfman, 1943). Instead of individual testing, which costs huge time and efforts, the soldiers were specifically divided into groups. The blood samples of each group were tested and if the test outcome for a group was positive, it was hypothesized that at least one soldier in the group was infected. Otherwise, all members of the group were healthy. More importantly, the test outcome could be used to exactly identify the infected individuals. This technique was known as *group testing*.

If a group testing strategy is applied with t test samples and N items, the strategy can be represented using a t x N binary matrix, $M = (m_{ij})$ where

$(m_{ij}) = 1$ iff item *j* belongs to test sample *i*. M_i is also used 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 *i*th test sample, and M^j is the set of tests containing 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 the first test sample which corresponds to first row in testing matrix.* M^1 *corresponds to the first column, indicates the set of tests containing the first item. In that case, only the first and the second 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 cover 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 an arbitrary column is selected, 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*, it can always be determined where they are from the test outcome using non-adaptive combinatorial group testing theory. Specially, if *M* is *d*-disjunct matrix, the positive items can be determined effectively using proper decoding algorithms.

Example 3. The f*ollowing 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
```

An assumption is made that 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 positive, 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.

Detecting bugs using group testing

This section discusses the use of group testing to determine fault locations in a program. Based on the authors' knowledge, this is the first time group testing is used to detect program's bugs although it has many applications in other fields (Clifford et al., 2010; Goodrich et al., 2005; Kainkaryam, 2009; Khattab et al., 2008; Rudra & Uurtamo, 2010; Xin et al., 2009). Firstly, a *unit block* of a program is defined - 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*. This path is *feasible* if there exists an input that makes *P* execute from B_i to B_n with the same order as described in P_C, otherwise it is *infeasible*.

Example 4. *Suppose we have a function:*

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

The unit-block representation will be:

```
if S0
  S1
else
  S2
```

Thus, this function has two execution paths: P1 = (S0, S1) and P2 = (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 test cases are used to test a program *P*, it can be considered as if testing matrix *M* is used, 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 error 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 structure.

Example 5. *If one can produce two test cases which correspond to P1 and P2 for the function in Example 4, the testing matrix will be:*

S0	S1	S2
1	1	0
1	0	1

Suppose the function has at most 1 bug. Since the testing matrix is 1-separable, the buggy block can be determined based on the test outcome. If the test outcome is (0, 0), the function does not have error. If the test outcome is (1, 1), the buggy block is S0. Similarly, if the test outcome is (1, 0) or (0, 1), the buggy block is S1 or S2 respectively[ii].

The experiments show that using group testing alone is quite good in finding bugs in the students programs, but sometimes wrong results were also possible (group testing does not return blocks contain buggy statements). In those cases, students do not know how to continue. So the researchers proposed another approach in which the spectrum-based fault localization and dynamic relevant slicing were combined to deal with this problem.

SLICING SPECTRUM-BASED FAULT LOCALIZATION

Spectrum-based fault localization

Fault localization methods aim to detect bugs in a program based on testing, the same goal as group testing, and can be divided into three categories: *spectrum-based*, *model-based*, and *proof-based*. Whereas model-based (Abreu et al., 2009; Abreu et al., 2008) and proof-based (Christ et al., 2013; Ermis et al., 2012; Jose & Majumdar, 2011) methods return the output similar to group testing output, spectrum-based ones are different. In these methods, the input is program executions with a predefined test suite, and the output are statements rankings from the most to the least suspicious. With such output, students are guaranteed to find bugs when they trace down through these rankings[iii]. Hence, this paper focuses on spectrum-based fault localization because it is a good complement for group testing. Some spectrum-based methods are presented as follows.

- Zhang et al. (2012) highlights a method that use 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 a line through them. The statement that has steeper line is more likely to contain errors.

- 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 alternation produces correct output. Those statements contain values that are more likely to produce correct output after the alternation are more likely the faults.

- In Abreu et al. (2007) and Jones & Harrold (2005), two methods called *Tarantula* and *Ochiai* are introduced, which build a function that maps each statement to suspicious scores. The statement with higher score is more likely the buggy one. The details of these functions are explained below.

- Renieris & Reiss (2003) present some other methods such as set-union, set-intersection, and nearest neighbor. 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.

In the above methods, Tarantula and Ochiai were chosen to be implemented in the system because of their effectiveness and efficiency. The functions used in these methods are:

- Tarantula: $s(e) = \dfrac{f(e)/tf}{p(e)/tp + f(e)/tf}$
- Ochiai: $s(e) = \dfrac{f(e)}{\sqrt{tf * (p(e) + f(e))}}$

in which:

- *s(e)*: the suspicious score of statement *e*
- *tp*: the number of passed test cases
- *tf*: the number of failed test cases
- *p(e)*: the number of passed test cases go through statement *e*
- *f(e)*: the number of failed test cases go through statement *e*

Example 6. *Suppose we have a function that returns absolute value of an integer as follow (statement in line 4 is buggy):*

```
1:  int abs(int n)
2:  {
3:    if (n > 0) {
4:      return -n;
5:    } else {
6:      return -n;
7:    }
8:  }
```

If we have two test cases n = 1 and n = -1, we will have the following summary table:

Statement in Line	*p(e)*	*f(e)*	*Suspicious score*	
			Tarantula	*Ochiai*
3	*1*	*1*	*0.5*	*0.7*
4	*0*	*1*	*1*	*1*
6	*1*	*0*	*0*	*0*
	tp = 1	*tf = 1*		

We can see both methods return statement in line 4 as the most suspicious statement.

Although these methods can give the good results (the buggy statements get high suspicious scores) in most cases, they have a shortcoming. Using these functions, statements in the same unit block always have the same suspicious scores although some statements do not affect the output. That lessens the quality of ranking statements. We can fix this problem by combining spectrum-based fault localization with slicing.

Slicing spectrum-based fault localization

Slicing are methods that can remove statements which do not affect the program output. Based on the work by When et al. (2011) and Zhang et al. (2005), the researchers have decided to combine dynamic relevant slicing with spectrum-based fault localization to get better result[iv].

Example 7. *The same function as in Example 6 is considered with one additional statement (line 4):*

```
1:  int abs(int n)
2:  {
3:    if (n > 0) {
4:      int i = 0;
5:      return -n;
6:    } else {
7:      return -n;
8:    }
9:  }
```

With test case satisfies condition n > 0, an execution path goes through statements in line 3, 4, and 5. Statement in line 4 does not have any effect on the program output. Using dynamic slicing, this statement can be removed from the above execution path.

To apply the slicing spectrum-based fault localization, the same functions are used with two modifications in calculating *p(e)* and *f(e)*. Firstly, the execution paths are sliced to filter out irrelevant statements. After that, *p(e)* and *f(e)* are calculated as the number of passed/failed test cases that have statement *e* in sliced execution paths.

Example 8. *With the function as in Example 7 and with the test suite {n = 1; n = -1}, if spectrum-based fault localization with original execution paths is used, statement in line 4 and 5 have similar suspicious score equals 1 (and this is the highest score). Whereas the highest score for the statement in line 5 is desirable, the score for the statement in line 4 is not needed at all. Using* *dynamic slicing, statement in line 4 is removed from the execution path and it will get suspicious score equals 0. In the end, the statement in line 5 is the only highest score statement.*

CASE STUDIES

In this section, two wrong implementations of bubble sort algorithm are analysed in detail. The right implementation is as follow:

```
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;
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 (right implementation)

Case study 1

```
1: int* sort(int n, int a[])
2: {
3:  int i = n - 1;
4:  while (i > 0) {
5:   int j = 0;
6:   while (j < i) {
7:    if (a[j] > a[j + 1]) {
8:     int temp = a[j];
9:     a[j] = a[j + 1];
10:    a[j + 1] = temp + 1;
11:   }
12:   j = j + 1;
13:  }
14:  i = i - 1;
15: }
16: return a;
17:}
```

Listing 2. The case study program (wrong implementation 1)

The statement in line 10 is logically wrong. Instead of `a[j + 1] = temp;` it is written as `a[j + 1] = temp + 1;`. Because that statement belongs to block S5, when using group testing, S5 is expected to be returned as error block. On the other hand, when analyzing with slicing spectrum-based fault localization, that statement should have the highest score.

The testing matrix for the above program has 1343 rows and 51 columns. In this matrix, some rows represent infeasible execution paths or, Test Case Generator module cannot generate test cases for them. Furthermore, some columns represent the same blocks because these blocks are repeated in loop structure. After deleting these rows and compacting each group into columns by representing the same blocks into one column, a testing matrix with only 11 rows and 9 columns left is produced.

$$\begin{array}{ccccccccc}
S0 & S1 & S2 & S3 & S4 & \mathbf{S5} & S6 & S7 & S8 \\
1 & 1 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{0} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & \mathbf{0} & 1 & 1 & 1
\end{array}$$

The test outcome after running the 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 test outcome is identical to S5-column, the system returns S5 as an error block.

Using the slicing spectrum-based fault localization, the statement in line 10 gets the highest suspicious score equals to 1.0 for both functions. The statements in lines 8 and 9 also have the same scores because they are in the same block with line 10 and are not removed by the slicing process. Hence, the results of the group testing and slicing spectrum-based fault localization are quite similar.

Case study 2

```
1: int* sort(int n, int a[])
2: {
3:  int i = n - 1;
4:  while (i > 0) {
5:   int j = 0;
6:   while (j < i) {
7:    if (a[j] < a[j + 1]) {
8:     int temp = a[j];
9:     a[j] = a[j + 1];
10:     a[j + 1] = temp;
11:    }
12:    j = j + 1;
13:   }
14:   i = i - 1;
15: }
16: return a;
17:}
```

Listing 3. The case study program (wrong implementation 2)

This time the wrong statement is in line 7. Instead of `if (a[j] > a[j + 1])` it is written as `if (a[j] < a[j + 1])`. Block S4 is expected to be returned when using group testing and statement in line 7 will have the highest score when using slicing spectrum-based fault localization.

Similar to Case Study 1, the final test matrix will have 11 rows and 9 columns as follow:

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 1 1 1 1 1 1 1 1 1). In this case, the test outcome is identical not only to S4-column but also S2, S3, S6, and S7-columns, so the system will return all of them as error blocks. We cannot avoid this result because in this program S2, S3, S4, S6, and S7 are always executed together, so their columns in test matrix are identical. But this result is not too bad, because when programmers want to debug, the maximum number of statements they need to check is 5 (the total number of statements in these blocks) instead of 11 statements in the whole programs.

The results from using the slicing spectrum-based fault localization are as follow: with Ochiai function for statements in block S2, S3, S4, S6, and S7 that have similar highest suspicious scores equals 1.0, with Tarantula function there are 3 more statements in block S5 with that score. This case study proves that the use of slicing spectrum-based fault localization with Tarantula function will return the worst result.

EXPERIMENTS

The system was experimented with 6 well-known algorithms as shown in Table 1.

In each algorithm, different implementations are created. Each implementation has exactly one bug. The experiments are used to test whether the system can show useful information to help programmers in the debugging process.

Example 9. *Below are two implementations of finding the absolute value algorithm with bugs.*

Implementation 1:

```
int abs(int n) {
  if (n >= 0) {
    return n;
  } else {
    return n;
  }
}
```

Implementation 2:

```
int abs(int n) {
  if (n >= 0) {
    return n + 1;
  } else {
    return -n;
  }
}
```

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	3
VI	Bubble sort	Sorting an array using bubble sort algorithm	4

To prove the usefulness of the system results, the number of statements programmers need to check is calculated in order to find the buggy one. These numbers are calculated from two contexts: programmers are lucky and programmers are unlucky as follow:

- Group testing:
    - Lucky: Among all the statements in returned blocks, the buggy statement is checked first.
    - Unlucky: Among all the statements in returned blocks, the buggy statement is checked last.

- Slicing spectrum-based fault localization:
    - Lucky: Programmers check the statement from high to low suspicious scores. If the buggy statement has similar score with others, it is checked first.
    - Unlucky: Programmers check the statement from high to low suspicious scores. If the buggy statement has similar score with others, it is checked last.

It is also noted that sometimes group testing returns wrong results. In these cases, the number of statements needs to be checked is defined as the number of statements in the whole program regardless of the context. In the experiments, there are 4 implementations (2 for algorithm I, 1 for algorithm III, and 1 for algorithm IV) that indicate that group testing does not give the right answer.

The results of the experiments are shown in Tables 2, 3, and 4. These tables indicate that by using group testing, programmers only need to check 1 to 2 statements if they are lucky. In case programmers are unlucky, the maximum number of statements they need to check is 8.33 in Selection sort algorithm. Similarly, the minimum number of statements programmers need to check while analyzing program with slicing spectrum-based fault localization is only 1 to 2 statements, while the maximum number are 9.67 and 9.33 (using Tarantula and Ochiai function respectively).

Table 2. The experiment results for group testing

Algorithms	**Average number of statements in implementations**	**Average number of statements need to be checked**	
		Lucky	**Unlucky**
Finding absolute value	3.14	1.57	1.71
Checking odd/even property	3.00	1.00	1.00
Finding maximum number	4.25	1.75	1.75
Calculating factorial	5.57	2.00	3.71
Selection sort	13.00	1.00	8.33
Bubble sort	12.50	1.00	4.50

Table 3: The experiment results for
slicing spectrum-based fault localization (Tarantula function)

Algorithms	**Average number of statements in implementations**	**Average number of statements need to be checked**	
		Lucky	**Unlucky**
Finding absolute value	3.14	1.43	1.43
Checking odd/even property	3.00	1.00	1.67
Finding maximum number	4.25	1.25	1.25

Calculating factorial	5.57	1.57	4.00
Selection sort	13.00	1.00	9.67
Bubble sort	12.50	2.00	6.25

Table 4: The experiment results for slicing spectrum-based fault localization (Ochiai function)

Algorithms	**Average number of statements in implementations**	**Average number of statements need to be checked**	
		Lucky	**Unlucky**
Finding absolute value	3.14	1.29	1.29
Checking odd/even property	3.00	1.00	1.00
Finding maximum number	4.25	1.25	1.25
Calculating factorial	5.57	1.57	3.71
Selection sort	13.00	1.00	9.33
Bubble sort	12.50	1.00	4.50

CONCLUSION

This paper presented a framework to help students practice their programming skills. Besides the ability to verify the correctness of the program, the proposed system can identify bugs automatically, thanks to group testing and slicing spectrum-based fault localization. However, the system still needs improvement. Firstly, group testing and spectrum-based fault localization cannot do well in case programs have more than one bug. To deal with this problem, a model-based fault localization method is offered. Secondly, often programs have bugs because of *missing code*, which means programmers forget to implement some cases during the specification. A method based on comparing the executions of students' programs and teacher's solution has been considered to detect the missing code. The results of the utilization of this method will be published in due course.

REFERENCES

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

Abreu, R., Zoeteweij, P., & van Gemund, A. J. (2008, July). An observation-based model for fault localization. In *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). ACM.

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2007, September). On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007* (pp. 89-98). IEEE.

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

Christ, J., Ermis, E., Schäf, M., & Wies, T. (2013, January). Flow-sensitive fault localization. In Giacobazzi, R., Berdine, J., & Mastroeni, I. (Eds.), *Verification, Model Checking, and Abstract Interpretation* (pp. 189-208). Springer Berlin Heidelberg.

Clifford, R., Efremenko, K., Porat, E., & Rothschild, A. (2010). Pattern matching with don't cares and few errors. *Journal of Computer and System Sciences*, *76*(2), 115-124.

Dorfman, R. (1943). The detection of defective members of large populations. *The Annals of Mathematical Statistics*, *14*(4), 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.

Ermis, E., Schäf, M., & Wies, T. (2012). Error invariants. In Giannakopoulou, D., & Mery, D. (Eds.), *FM 2012: Formal Methods* (pp. 187-201). Springer Berlin Heidelberg.

Goodrich, M. T., Atallah, M. J., & Tamassia, R. (2005, January). Indexing information for data forensics. In Ioannidis, J., Keromytis, A., & Yung, M. (Eds.), *Applied Cryptography and Network Security* (pp. 206-221). Springer Berlin Heidelberg.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010, October). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research* (pp. 86-93). ACM.

Jeffrey, D., Gupta, N., & Gupta, R. (2008, July). Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis* (pp. 167-178). ACM.

Jones, J. A., & Harrold, M. J. (2005, November). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 273-282). ACM.

Jose, M., & Majumdar, R. (2011, June). Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Notices* (Vol. 46, No. 6, pp. 437-446). ACM.

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.

Kainkaryam, R. M., & Woolf, P. J. (2009). Pooling in high-throughput drug screening. *Current opinion in drug discovery & development*, *12*(3), 339.

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

Khattab, S., Gobriel, S., Melhem, R., & Mossé, D. (2008, April). Live baiting for service-level dos attackers. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE* (pp. 171-175). IEEE.

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 Escalona, M. J., Cordeiro, J., & Shishkov, B. (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, August). A framework for automatic verification of programing exercises. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on* (pp. 41-45). IEEE.

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

Rudra, A., & Uurtamo, S. (2010). Data stream algorithms for codeword testing. In Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., & Spirakis, P. (Eds.), *Automata, Languages and Programming* (pp. 629-640). Springer Berlin Heidelberg.

Wen, W., Li, B., Sun, X., & Li, J. (2011). Program slicing spectrum-based software fault localization. In *SEKE* (pp. 213-218).

Xin, X., Rual, J. F., Hirozane-Kishikawa, T., Hill, D. E., Vidal, M., Boone, C., & Thierry-Mieg, N. (2009). Shifted Transversal Design smart-pooling for high coverage interactome mapping. *Genome research*, *19*(7), 1262-1269.

Zhang, X., He, H., Gupta, N., & Gupta, R. (2005, September). Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (pp. 33-42). ACM.

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

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

[ii] Sometimes, the test outcome has positive values but it is not identical with any column in test matrix. In these cases, we return all columns cover the test outcome.

[iii] Slicing spectrum-based fault localization can also return wrong answers if test cases are not good enough to detect failed execution paths. In our experiments, we do not encounter this problem, thanks to Test Cases Generator module.

[iv] Actually, we can also combine group testing with slicing but we do not do so because we want to keep group testing as simple as possible.