ISBN 978-989-9121-49-2
61st KUALA LUMPUR International Conference on Research in "Science, Engineering & Technology" (KRSET-25)
Kuala Lumpur (Malaysia) June 2-4, 2025

A Study of Integer Input Test Data Generation for Validating Basic C Programs

Zhikang Li[1], Xiqin Lu[2]

[1]Department of Information and Communication Systems, Okayama University, Japan
[2]College of Information Science and Engineering, Ritsumeikan University, Japan

Abstract**:** *In many universities around the world, C programming is taught in basic programming or software courses. To assist self-studies of novice students, we have developed a C programming learning assistant system (CPLAS). CPLAS provides code writing problem (CWP) that asks students to write source codes that will generate the expected outputs to given inputs. To assist teachers at marking the submitted codes, we have implemented the code validation function that automatically checks their correctness through compiling codes, running them, and comparing outputs with expected ones. However, since the input and output data are manually generated and the test data is limited, it may miss incomplete codes. It is necessary to dynamically generate different test data to avoid it. In this paper, we present a test data generation to automatically generate new test data for source codes that accept integer inputs. For evaluations, we used source codes for 10 C programming assignments submitted from first-year undergraduate students in a course and confirmed the correctness of the proposal.*

Keywords: *Test Data Generation, C Programming, Code Validation.*

1. Introduction

Presently, C programming is taught in basic programming courses to undergraduate students in a lot of universities around the world. C programming is useful for studying advanced and practical programming languages, which are more suitable and easier to implement practical and large-scale application systems. Besides, C programming can be used to teach fundamental data structure and algorithms, and computer architecture for students in information technology or computer science departments, which are essential subjects in IT/CS curriculums.

To assist its self-studies by C programming, we have developed the *C programming learning assistant system* (*CPLAS*). *CPLAS* provides several types of exercise problems for C programming beginners to solve by self-studies. Among them, *CPLAS* provides *code writing problem (CWP)* that asks students to write source codes that will generate the expected outputs to the given inputs. To assist teachers at marking the submitted source codes from a lot of students, we have implemented the code validation function using Python that automatically checks their correctness through compiling codes, running them, and comparing outputs with expected ones.

This function validates the correctness of the source codes through the three tests: 1) compiling test, 2) running test, and 3) output test [1]. The compiling test compiles each source code using the *gcc* compiler. If passed, the running test executes the compiled code. If passed, the output test checks whether it outputs the correct data or messages with the correct format for the given input data that are given by the teacher.

Unfortunately, the input and output data are manually generated by the teacher for each CWP assignment so that the test data is limited in the previous study. As a result, it may miss incomplete source codes that may

output the fixed ones. Therefore, it is necessary to automatically and dynamically generate different test data during the validation process to avoid it.

In this paper, we present a test data generation to help the teacher generate integer input test data of C source codes automatically. Besides, three patterns of the standard input `scanf` function for code structure are considered: 1) `scanf` functions without loops, 2) `scanf` functions in one loop, and 3) `scanf` functions in different loops. The proposed method follows the four steps: 1) it reads the number of test cases from the model source code given by the teacher, 2) it randomly generates a set of integer input data within the fixed range, 3) it compiles and runs the model source code with the generated input data to get the output data, and 4) makes the input and output data files as the test data.

For evaluations of the proposed method, we applied it to 10 source codes of 10 C programming assignments by first-year undergraduate students in Nihon university, Japan, and confirmed the correctness of the proposal.

The rest of this paper is organized as follows: Section 2 discusses related works in literature. Section 3 presents the proposed test data generation method for C programming. Section 4 evaluates the proposal through applications to 10 source codes. Finally, Section 5 concludes this paper with future work.

2. Related Works

In In this section, we briefly review related works in literature.

In [2], Girgis presented an automatic test data generation technique that uses a *genetic algorithm (GA)*. It is guided by the data flow dependencies in the program, to search for test data to cover its def-use associations. The GA conducts its search by constructing new test data from previously generated test data that are evaluated as effective test data. This approach can be used in the test data generation for programs with/without loops and procedures. It accepts as input an instrumented version of the program to be tested, the list of def-use associations to be covered, the number of input variables, and the domain and precision of each input variable. While our study focuses on C programming whose inputs need to be received by a user from the console manually, and the generation of automatic integer inputs.

In [3], Fraser et al. presented implementation results and discussions of a tool *EVOSUITE* that is applied to 100 Java projects. *EVOSUITE* is a research prototype, which automatically generates unit test suites for classes written in Java. Users can the generated unit test suites to test if a Java source code works correctly. On the other hand, our proposal validates a C source code by comparing the automatically generated input/output, running and compiling tests.

3. Integer Test Data Generation

In this section, we present the integer test data generation method for CPLAS.

3.1. Overview

When a C source code under considerations contains `scanf` functions, the input data to this program needs to be entered from the console by a user manually. In the code validation function, to validate source codes submitted by students, the test data generation method is proposed to automatically generate integer input data and the corresponding output data for testing a C source code.

3.2. Three patterns

The test data generation method considers the following three types of source codes for `%d` integer input.

3.2.1. Pattern 1: %d Input without loops

Figure 1 shows a source code for the "*Compare two integers entered from the console"* assignment. This source code has two `scanf` functions at line 8 and line 10, where they are not included in any `for`, `while`, or `do while` loop. For this code, the proposed method randomly generates two integer numbers between -20 and 20

and saves them to the input data file, as shown in Figure 2. Then, it compiles the source code, runs the program using the generated input data, and saves the output data of the program to the output file, as shown in Figure 3.

In this example, we set the number of test cases N with N=2, so that two different groups of input data are generated in the input data file and the corresponding output data are saved in the output data file.

```c
#include <stdio.h>

int main(void) {
   int a, b;

   printf("Please enter two integers\n");
   printf("Integer A:");
   scanf("%d", &a);
   printf("Integer B:");
   scanf("%d", &b);

   if (a == b) {
      printf("Both are %d\n", a);
   } else if (a > b) {
      printf("The larger value is %d\n", a);
      printf("The smaller value is %d\n", b);
   } else {
      printf("The larger value is %d\n", b);
      printf("The smaller value is %d\n", a);
   }
   return 0;
}
```

Fig. 1: C source code: compare two integers.

```
Input example 1
9
7

Input example 2
-10
2
```

Fig. 2: Generated input data file.

```
Execution example 1
Please enter two integers
Integer A:9
Integer B:7
The larger value is 9
The smaller value is 7

Execution example 2
Please enter two integers
Integer A:-10
Integer B:2
The larger value is 2
The smaller value is -10
```

Fig. 3: Generated output data file.

3.2.2. Pattern 2: %d Input in one loop

Figure 4 shows a source code for the "*Output the total value among the three integers input from the console"* assignment. The source code has only one `scanf` function at line 8, where it is included in a `for` loop at lines 6 - 10. For this code, the proposed method counts the number of the `for` loop first, and then, it determines the number of input data. In this code, since the `for` loop repeats the procedure three times, it generates three `%d` input data automatically. Here, the data are also randomly generated in the range between -20 and 20. It is noted that this range can be adjusted according to actual conditions. Figure 5 shows an example of the generated input data for N=1, the three integers, 4, 10, 5, are generated. Figure 6 shows the corresponding output data.

```c
#include<stdio.h>

int main(){
    int x, sum=0, i;

    for(x=1; x<=3; ++x){
        printf("Enter #%d: ",x);
        scanf("%d",&i);
        sum = sum + i;
    }
    printf("Total Sum of the three numbers is %d\n",sum);
}
```

Fig. 4: C source code: output the total integer value.

```
Input example 1
4 10 5
```

Fig. 5: Generated input data file.

```
Execution example 1
Enter #1: 4
Enter #2: 10
Enter #3: 5
Total Sum of the three numbers is 19
```

Fig. 6: Generated output data file.

3.2.3. Pattern 3: %d Input in different loops

Figure 7 shows the source code for the "*Input four positive integers from the console and output the max value"* assignment. The source code has two `scanf` functions at line 8 and line 12, where one `scanf` function is included in the `for` loop at lines 6 - 16 and another is included in the `while` loop at lines 9 - 13. If a negative integer is entered from the console, it is requested to re-enter an integer until four positive integers are entered. The number of input data is not only determined by the number of `for` loop, but also by the data entered by the user on the console.

For this code, the proposed method confirms both the number of `for` loops and the condition of the `while` loop, and then, determines the number of input data. Figure 8 shows an example of the generated input data for N=1, where seven integers include four positive, and three negative integers are automatically generated. Figure 9 shows the corresponding generated output data.

```
1  #include <stdio.h>
2
3  int main(void) {
4      int i, max, value;
5
6      for (i = 1; i <= 4; i++) {
7          printf("Value No.%d:", i);
8          scanf("%d", &value);
9          while (value <= 0) {
10               printf("Value is not a positive number\n");
11               printf("Value No.%d:", i);
12               scanf("%d", &value);
13         }
14         if (value > max)
15             max = value;
16     }
17     printf("The maximum value is %d\n", max);
18
19     return 0;
20 }
```

Fig. 7: C source code: output the max integer value.

```
Input example 1
10 -10 2 17 -13 -1 13
```

Fig. 8: Generated input data file.

```
Execution example 1
Value No.1:10
Value No.2:-10
Value is not a positive number
Value No.2:2
Value No.3:17
Value No.4:-13
Value is not a positive number
Value No.4:-1
Value is not a positive number
Value No.4:13
The maximum value is 17
```

Fig. 9: Generated output data file.

3.3. Algorithm procedure

The algorithm procedure is described as follows:

1. Read the number of test cases N that will be entered from the user and initialize the number of generated test cases by k=0.
2. Read the C source code and find the `scanf` functions in the source code. If no `scanf` function is found, without generating input data, compile and run the program, and save the output data into the output data file. Then, terminate the procedure. Otherwise go to Step 3.
3. Find the `for/while/do while` in the source code. If no `for/while/do while` is found or all the `scanf` functions are included in one `for/while/do while` loop, generate the input data with random numbers between -20 and 20, save them to the input data file. It is noted that the number of generated input data is determined by the number of loops. Then, go to Step 5. Otherwise, go to Step 4.
4. When the `scanf` functions are included in the different `for/while/do while`, generate the input data with random numbers between -20 and 20, save them to the input data file. In this case, the number of generate input data is determined both by the number of loops and by the generated input data.
5. Compile the C source code, run the program with the generated input data file, and save the output data of the program into the output data file.
6. Terminate the procedure if k=N. Otherwise, increment k by 1 and go to Step3.

Figure 10 shows the algorithm flowchart of the proposed method.

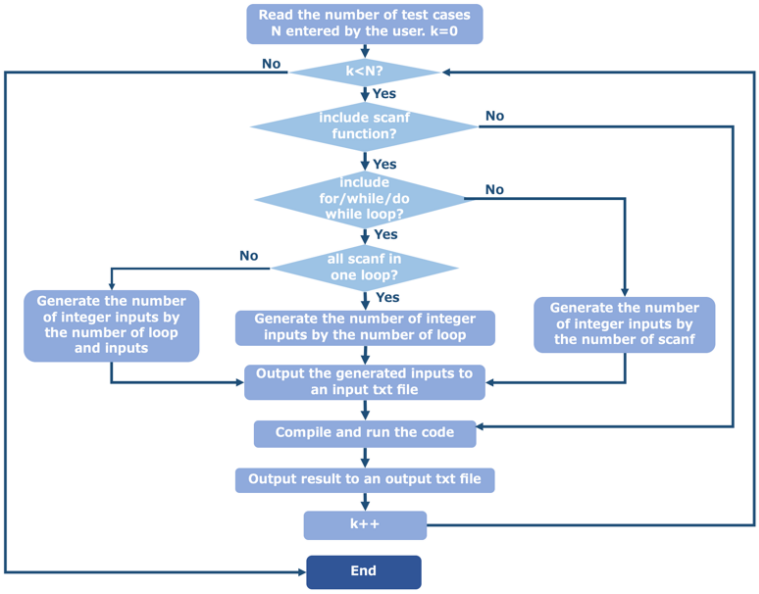

Fig. 10: Algorithm flowchart.

4. Evaluation

In this section, we evaluated the proposal through applications to the source codes for 10 assignments of a C programming course to first-year undergraduate students in Nihon university. Table 1 shows the ID, the description for each source code, the number of `scanf` functions in the source code, the number of `for`/`while`/`do while` loops in the source code, and the pattern type in Section 3.2. The source codes with ID = 1, 2, and 3 in the Table 1 correspond to the three examples given in Section 3.2. Besides, we tested the source code with ID = 4 without a `scanf` function, as shown in Table 1.

In our experiments, we compiled and run each C program manually using the input data generated by the method and compared the output data with the console output. It was found that all the results were the same. Therefore, it was confirmed that the proposed method can successfully generate the correct integer input and output data files for each source code automatically.

TABLE I: TESTED SOURCE CODES

ID	Description of source code	Number of scanf in the source code	Number of loops in the source code	Pattern type
1	Compare two integers entered from the console	2	2	1
2	Output the total value among three integer inputs	1	1	2
3	Output the max value among the four integers input if a negative integer is entered, re-enter it	2	2	3
4	Output a multiplication table	0	2	No input
5	Determine whether an integer input is odd or even	1	0	1
6	Output the number of students/scores based on inputs	2	1	2
7	Output the month now according to the integer input	2	2	2
8	Output size comparison results according to integer inputs	2	1	3
9	Output the information of oranges and apples	2	0	1
10	Output the values of an array based on inputs	2	2	3

5. Conclusion

This paper presented an automatic test data generation method to automatically generate integer inputs and execution outputs for C programming programs. The applications to 10 C source codes with three integer input patterns confirmed the correctness and effectiveness of the proposal. In future works, we will extend the method to other input data types such as double/char/String, import this method into the code validation function, and import the function into the CPLAS platform for self-studies by C programming beginners.

6. References

[1] A. A. Puspitasari, N. Funabiki, X. Lu, H. Qi, H. H. S. Kyaw, and K. Ueda, "An implementation of code validation function in C programming learning assistant system," *International Journal of Learning and Teaching (IJLT)*, vol. 9, no. 1, pp. 24-30, 2023.

https://doi.org/10.18178/ijlt.9.1.24-30

[2] Moheb R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm," *Journal of Universal Computer Science*, vol. 11, no. 6, pp.898-915, 2005.

[3] Fraser Gordon, and Andrea Arcuri. "Evosuite: On the challenges of test case generation in the real world." in *IEEE Proc. Int. Conf. soft. Testing, validation*, pp.363-396, 2013.

https://doi.org/10.1109/ICST.2013.51