PARAMETRIC CODING ASSIGNMENTS

By

Bei Pang

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Steven S. Lumetta

December 2016
Abstract

This thesis introduces a set of parametric coding assignments for ECE120, an entry-level course of the Department of Electrical and Computer Engineering. These coding assignments are called “parametric” because the function implemented in these assignments varies according to different input values. Another significant feature of the parametric coding assignments is that the assignments of different students are different from each other in terms of parameters and implementation requirements. But the difficulties and skills required to complete an assignment for all students are identical. These coding assignments use an automated feedback framework to grade students’ codes with KLEE, a symbolic testing tool, and to provide feedback (grading results and error messages) on students’ code submitted via Subversion, version control software used to distribute and collect coding assignments. The automated feedback framework was designed and implemented by Jianxiong Gao, a Ph.D. student of Prof. Steve S. Lumetta, and this framework was successfully applied to ECE220 in previous semesters.

In this semester, Fall2016, around one-quarter of ECE120 students signed up voluntarily to try these assignments. However, because these coding assignments are optional exercises for the students, we received only a small number of submissions. Discussion of the limited but enlightening submissions is in Chapter 4.

Subject Keywords: coding assignment; automated feedback; concolic testing
Acknowledgments

Special thanks to Jianxiong Gao, a Ph.D. student of Prof. Steven S. Lumetta, who offered me valuable ideas when I was designing these parametric coding assignments.
# Contents

1. Introduction .......................................................................................................................... 1  
   1.1 Motivation ......................................................................................................................... 1  
   1.2 Background ....................................................................................................................... 3  
      1.2.1 Automated Feedback Framework .............................................................................. 3  
      1.2.2 KLEE and Concolic Testing .................................................................................... 3  
2. Literature Review .................................................................................................................. 6  
3. Description of Research Results ......................................................................................... 7  
   3.1 Components of a Parametric Coding Assignment ............................................................... 7  
      3.1.1 Assignment Specifications ......................................................................................... 7  
      3.1.2 Assignment Generator ............................................................................................. 7  
      3.1.3 Processing and Grading Components ......................................................................... 9  
   3.2 Design and Implementation of Parametric Coding Assignments ....................................... 10  
      3.2.1 Implementing a Multi-Region Function ...................................................................... 11  
      3.2.2 Printing Canonical SOP / POS .................................................................................. 15  
      3.2.3 Printing a K-map ........................................................................................................ 17  
      3.2.4 Bit-sliced Logic Simulation ......................................................................................... 19  
      3.2.5 Finding the Stable States of a Sequential Feedback Circuit ....................................... 21  
   3.3 Result of Production Run in Semester Fall 2016 ................................................................. 23  
4. Conclusion ............................................................................................................................ 26  
References .................................................................................................................................. 27  
Appendix A: Feedback messages of each assignment ............................................................... 28  
   A.1 Error Conditions and Messages of Week3 (Multi-region function) ............................... 28  
   A.2 Error Conditions and Messages of Week4 (Printing Canonical SOP/POS) .................. 31  
   A.3 Error Conditions and Messages of Week5 (Printing K-map) ......................................... 33  
   A.4 Error Conditions and Messages of Week6 (Simulating Bit-sliced Logic) .................... 36  
   A.5 Error Conditions and Messages of Week7 (Finding the Stable States) .......................... 38
1. Introduction

This set of five coding assignments is called “parametric” because each of these assignments have a tuple of parameter values determining the specific assignment for any given student. Another feature of the parametric coding assignment is that the assignments of two students are different from each other in terms of parameters and implementation requirements, although the difficulties and skills required to complete an assignment for all students are similar. To introduce variance between individuals and to guarantee consistency for each student, the first step is to generate a unique seed for each student. Using a sequence of random numbers generated by the seed, a unique assignment and a correct solution can be created. The parametric coding assignment is developed on top of the automated feedback framework built by Jianxiong Gao and Bei Pang in 2015. An introduction to this framework appears in Section 1.2.1.

This thesis emphasizes the design and implementation of each assignment. These five parametric coding assignments were available to ECE120 students as optional exercises this semester (Fall 2016). These five assignments were distributed in a weekly manner: the students have one week to complete an assignment, and are not able to receive feedback messages after the assignment’s deadline.

1.1 Motivation

The number of students enrolled in programming classes increases as the software industry become more and more successful and well-known to the public. In contrast, the number of teaching staff in programming classes does not increase appreciably. Therefore, in the computer laboratories of these programming classes, it is common to have only one or two teaching assistants helping with students in the entire laboratory. J. Gao et al. [1] argues that feedback about programs plays a significant part of improving programming skills, especially for students in entry-level programming courses, but a small
teaching staff cannot provide prompt personalized feedback for individuals, which limits the quality of education. To solve this problem, the automated feedback framework was developed to provide feedback messages to students within minutes after submitting their code.

The lack of teaching staff may trigger an even more serious problem. When a student cannot receive desired help from teaching staff, he/she tends to seek help from friends. But his/her friends are not always experienced enough to help him/her understand the material; instead, they may simply give out the answers. As a result, the student may only know the answer to a single question but not be able to fully understand the related material. With parametric coding assignments, we strived to address this issue by making the assignment of each student unique. A student may still benefit from understanding how a friend solve their problem, but the student must still master that knowledge in order to solve their version of the same problem.

The assignments are designed to help ECE120 students get comfortable with C syntax so that the mechanics of programming are less of a stumbling block for them in ECE220. The content of these coding assignments is designed to help ECE120 students review and understand what they learned in class. Although ECE120 is not a programming class, completing coding assignments can significantly help students understand course materials by proactively applying these materials to solve problems. Because coding related questions can effectively facilitate the learning of the course material, coding related problems have already been introduced into many assignments of ECE120.
1.2 Background

1.2.1 Automated Feedback Framework

Most of today’s programming classes use traditional auto-graders to grade the students’ code several times a day or provide students with some test cases so that they can test by themselves. These methods cannot efficiently provide feedback to students’ code or comprehensively test students’ submissions. This automated feedback framework, however, can provide feedback within minutes after students’ submissions.

Programming courses at the University of Illinois at Urbana-Champaign use a software package called Subversion to distribute and collect coding assignments. The automated feedback framework has a Ruby script that constantly checks for new submissions in the course directory in Subversion. It looks up the activities of each student every 30 seconds, and whenever it detects a newly submitted code, this script initiates the grading process by executing an orchestration script of the grading components. This grading script first makes a copy of student code and modifies the copied code for grading purposes. Then the script compiles and executes the testing code with KLEE and collects the execution results to generate feedback and error messages.

1.2.2 KLEE and Concolic Testing

Unlike traditional auto-graders which require a human to design test cases for grading, the automated feedback framework uses KLEE, a concolic testing tool, to automatically generate test cases that maximize code coverage. Concolic testing is the combination of concrete value and symbolic execution. Symbolic execution described by Cadar et al. [2] is a software verification method used to automatically generate test inputs that can achieve a high code coverage. Generally speaking, unlike regular (concrete) execution that runs the code with randomly or manually constructed inputs, symbolic execution tools
run code with symbolic inputs. In addition, special operations that can manipulate these symbolic values are required. During a symbolic execution, when a program executes a conditional statement that contains symbolic variable(s), the tools follow both branches concurrently and maintain a set of conditions for each execution path. In order to cover a section of code, a test input that follows an execution path can into that section be generated by solving the current path condition.

Concolic testing tools combine concrete values into symbolic executions. The tools first randomly assign a concrete value to each input variable and executes the code. The tools expand the code coverage by trying to explore more execution paths: the tools negate one symbolic constraint in a path and solve the constraints for a new set of concrete values. This new set of values will then direct the program into a different execution path. The process continues until no more execution paths can be explored.

Figure 1.1 is an example of concolic execution. The tree-like graph on the right represents the execution paths of the code on the left. Concolic testing tool first assigns arbitrary values to inputs. In this example, x=0, and y=0. With these inputs, the execution of the code ends at line 3 and a set of inputs of x=0 and y=0 is generated. To explore more paths, the tool looks at the condition of this path and sets x=100000. Then, the second test with x=100000 and y=0 is generated. Finally, the tool solves a set of inputs that have x less than z. Since the tool records that z is twice y, the tool then sets y to a value such

```c
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) {  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```

---

Figure 1.1  Example of concolic testing. The orange line denotes the second execution path.
that $x$ is less than $2y$, for example, $y=50001$. As a result, the tool generates three sets of test inputs that can cover all the branches in the code.
2. Literature Review

The motivation behind developing the parametric coding assignments is to develop a set of coding assignments that can provide prompt feedback messages to facilitate the learning of ECE120’s course material. The automated feedback framework built by Jianxiong provided solution to this problem by successfully integrating concolic testing tool into assignment grading. The paper “Automated Feedback Framework for Introductory Programming Courses” [1] demonstrated how the framework uses KLEE, a concolic testing tool, to grade students’ coding assignments and how good the framework is in finding bugs missed by traditional grader. The concolic testing tool used by the automated feedback framework is called KLEE. KLEE is built by Cristian Cadar, Daniel Dunbar, Dawson Engler from Stanford University. In their paper “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs” [2], they explained the usage and architecture of KLEE and the astonishing result of using KLEE as a bug finding tool.

In general, the parametric coding assignments are implemented on the top of the automated feedback framework. It is the concolic testing tool used by the feedback framework that allows the possibility of grading unique assignments of students because it can generate test inputs automatically. The designing a parametric coding assignment includes: deciding the topic of the assignment, drafting and reviewing of the assignment specifications and skeleton code, and discussing the implementations of some important components. The design of the assignments was accomplished with the help of my advisor. The implementation processes consisted of implementing the assignment generator script, implementing the grading code that is responsible for the execution of KLEE, and modifying the grading script to detect the error and generate feedback messages. The error conditions and the corresponding feedback message are in Appendix A.
3. Description of Research Results

3.1 Components of a Parametric Coding Assignment

3.1.1 Assignment Specifications

Just like regular coding assignments, the parametric coding assignments have assignment specifications, documents that explain information and specify the requirements of each assignment. For an assignment that has a complex topic, the assignment specification may provide information and hints that help students complete their assignments. The assignment specifications are usually the primary resource for students to figure out the purpose of assignments and the work they should do. Therefore, reviewing the specification is always a significant task. The assignment specification of a parametric coding assignment has two parts: one PDF file that introduces the assignment and explains the implementation requirements, and one text file that specifies the unique part of the assignment for each student. It is possible to generate one PDF file that contains all the information, but that requires extra work of using LaTeX. Therefore, for simplicity, one extra text file is used instead. This text file is generated by the assignment generator discussed in the next session.

3.1.2 Assignment Generator

The assignment generator generates a unique assignment for each student. The assignment generator is a Ruby script that first reads a random seed, then uses the seed to randomly specify the elements that vary between students’ assignments. The random seeds obtained by hashing students’ netids are assigned to the students at the beginning of a semester. This script uses integers generated by this seed as the value of parameters (for example, the number of variables) or to select from potential options (for example, selecting the names of variables from the alphabet). The script generates the student-specific portion of the assignment specification (the text file) since it has all the elements (variables and choices) that define a unique assignment. The
script is also responsible for creating student-specific gold code (the solution to a coding assignment) for each student’s unique assignment. Despite each student’s assignment being unique, the general requirements are identical. As a result, the solutions of all of the students’ assignments share the same structure, and the differences exist only in several statements. Based on this observation, we develop skeleton code as a starting point to generate the gold code for the unique coding assignments. With the general structure written down, the skeleton code leaves some statements empty, and the assignment generator generates gold code by completing those empty statements with all the elements that define a unique assignment. Figure 3.1 shows code from the skeleton code of the first assignment. This thesis discusses the details of skeleton gold

```c
4   int main()
5   {
6    int N = <n>;   // Number of inputs
7    int x, y;     // Inputs from console
8    int out = 0;
9
10   // Print a prompt of reading N integers and read inputs
11   printf("Please input %d integers.\n", N);
12   scanf("%d%d", &x, &y);
13
14   // Implement the function specified in mp_spec.txt
15   <condition1> { 
16     out = <expression1> ;
17   }
18   <condition2> {
19     out = <expression2> ;
20   }
21   else{
22     int i = 0;
23     for(i = 1; i <= <bound>; i++){
24       out += <expression3> ;
25     }
26   }
27
```

Figure 3.1  Code snippet from a skeleton gold code. The brackets “<” and “>” represent the statements that should be changed by the assignment generator to make this code a valid gold code. The strings between the brackets are used to distinguish the statements.
code and further information about the assignment generator in section 3.2 in which the details and implementations of each parametric coding assignment are explained.

3.1.3 Processing and Grading Components

This part of a parametric coding assignment serves to grade and provide feedback to students’ submissions and technically belongs to the automated feedback framework. The processing component is a Ruby script that checks new submissions of students. The grading component consists of one orchestration Ruby script and other files for KLEE execution, assignment grading, and feedback generating. Files in the grading component are significantly different between different parametric coding assignments.

In order to grade students’ submissions, the grading component has to modify students’ code. The changes include changing the function name, “main”, in student code, and substituting fake “printf” and “scanf” functions in place of “printf” and “scanf”. Since ECE120 does not introduce function calls in C programming, all of the implementations of a student are in the “main” function. To execute the student code with KLEE, the grading component has to modify the student code such that the grading code can execute the students’ main function with a function call. Because student code uses “scanf” to read inputs, the “scanf” statements are modified to read symbolic inputs for concolic testing. Since the grading of parametric coding assignments relies on the output of “printf” statements, substituting the “printf” statements is necessary. Because these changes are required, the grading component always
works on copies of student’ submissions to prevent making changes to the actual code. Figure 3.2 is an example of modified student code.

```c
int practice(char* input_str)
{
    int x, y, j, i=1, out=0;
    practice_printf("Please input 3 integers: ");
    practice_scanf(input_str,"%d, %d, %d", &x, &y, &j);
    if((21*x)-(9*y)<0){
        out=(14*y)+((-14)*x);
        practice_printf("%d\n", out);
    }
}
```

Figure 3.2 Code snippet of modified student’s submission. The function name “main” was changed to “practice”. Both “printf” and “scanf” statements were replaced with function calls in the grading component.

### 3.2 Design and Implementation of Parametric Coding Assignments

This chapter discusses the design and implementation of the parametric coding assignments. There are a total of five parametric coding assignments, and each of the following subsections introduces a parametric coding assignment. The introduction of each assignment includes the content of the assignment, the required skills for the assignment, and the variance between different students’ assignments. The implementation of assignment generators, skeleton gold code, and ideas behind the design of the assignments are also presented.

All of the five parametric coding assignments require students to implement the main function of a C program. An incomplete version of the program is given to the students who must complete the code by implementing the “main” function. In general, the “main” function first uses a “printf” statement to print a prompt that asks for user inputs and then uses a single “scanf” statement to read the user inputs. After reading the inputs, students should implement the required functionality of the program. The
program is different for different students and specifications of this function depend on both the PDF and text assignment specifications. The program is designed to use input values to vary code paths and the results such that KLEE can help students find bugs more effectively because KLEE is able to solve input vector for each execution path. As for the traditional assignment grader, it would require a lot of human work to develop test cases that can cover all of the execution paths. Finally, the “main” function should print the output of the implemented function F using a single “printf” statement.

3.2.1 Implementing a Multi-Region Function

In this assignment, each student is asked to read the values for 2, 3, or 4 integer variables. The first two variables (x and y) are the coordinate of a point in a two-dimensional plane. Three constraints divide the plane into eight regions: a circular constraint (circle C, described by equation \( x^2 + y^2 = r^2 \)) whose center is at the origin, a linear constraint (line G, described by \( ax + by = 0 \)) passing through the origin, and another linear constraint (line H, described by \( cx + dy = e \)) not passing through the origin, but crossing through the circle C such that it intercepts the first linear constraint (line G) inside the circle C. All the above coefficients (a, b, c, d, and e) are integers. The function F takes the point \((x, y)\) as its input and outputs a value according to the position of the input point in the divided two-dimensional plane. The function F uses two conditional statements to determine in which region a point \((x, y)\) lies. The condition may be, for example, “above the line G” and “not inside circle C”. The outputs of different regions are calculated by one of two possible functions: linear polynomials of some of the input variables (for example, \(-68x + 3y - 74i\)), and summation functions that sum over a variable \(i\) from 1 to the absolute value of the last input variable of the cube of \(i\) multiplying another input variable. For example, if the input variables are just \(x\) and \(y\), the summation should be \(\sum_{i=1}^{abs(y)} i^3 * x\). In general, students must write conditional statements to determine in which region the input point lies and to print out the output of the function F.
The number of input variables, the constraint that divided the plane, the conditions to locate the region of the input point, and the output functions are different for each student and are specified in the student-specific portion of the assignment specification. Besides the specifications, each student receives a graph that describes what their constraints look like. This graph is generated by the assignment generator by modifying a gnuplot script. Figure 3.3 shows an example of such a graph.

When designing a parametric coding assignment, an important design decision is to determine the number of and the range of symbolic input variables that are used in the grading component. The symbolic input variables correspond (by design) to the values read from the keyboard with “scanf”. The number and range of symbolic inputs need to be tuned because the execution time of concolic testing largely depends on the number of conditional statements that branch on symbolic variables, and it is crucial to ensure that the execution time of grading ends in an acceptable time (usually 5 minutes). In
this assignment, the number of inputs is 2 to 4, which is fairly small. The assignment generator ensures that two linear constraints intersect within a circle of radius of three unit lengths and the radius of the circular constraint circle C is four to eight unit lengths larger than the radius of the intersection, and therefore, a small range of input variables is enough to generate test cases to maximize code coverage.

Another equally important design decision is to make sure that variance between assignments of different students is large enough for each student to have a unique assignment. However, it is difficult to guarantee the uniqueness of assignments in the real world; the design only makes sure that the number of possible different assignments is as large as possible so that the possibility of the students with same assignment being known to each other is fairly low. In this assignment, variation includes the three constraints, the conditions used to identify the region of the input point, and the calculation of output functions. The last two can be implemented by simply using random numbers to select from potential options. Constraint selection is more complicated, as we want to ensure the existence of integer-valued tests that differentiate correct implementations from incorrect ones. The discussion in the following part of this session focuses on the generation of three constraints.

The line G (described by equation $ax + by = 0$) is generated by assigning $a = M \times N$ and $b = M \times P$, where $M, N,$ and $P$ are three randomly generated integers. The resulting equation, $MNx + MPy = 0$, has an integer solution: $x = P$ and $y = -N$. Having an integer solution for the equation ensures that the correctness of assignments that have conditions such as “point $(x, y)$ lies not below line G” is verifiable. As for generating line H and circle C, since line H intersects line G within the circle C, one convenient approach is to first randomly generate the intersection point of line G and line H and then generate the circle C based on the intersection. Line H (described by $cx + dy = e$) can be generated by an approach similar to that for generating line G; by randomly choosing four integers $X, Y, N,$ and $P$
and assigning them to coefficients of the equation of Line H such that \(c = N, d = P,\) and \(e = N \times X + P \times Y.\) The resulting equation \(N \times x + P \times y = N \times X + P \times Y\) shows that the equation also has an integer solution: \(x = X\) and \(y = Y.\) Since circle C is defined only by its radius, it can be generated by randomly selecting a radius (integer) larger than the distance between the intersection point and the origin. Finally, the assignment generator needs to make sure that line G and line H are not parallel and that the acute angle between them is not too small to have one integer solution in all of the eight regions. If the generated constraints cannot fulfill these requirements, then the assignment generator starts the generation process over. Figure 3.4 is the code snippet of the assignment generator that generates the three constraints.

```java
loop do
    x = (prng1.rand(1..$x_bound) * (prng1.rand(0..1) * 2 - 1))
    y = (prng1.rand(1..$x_bound) * (prng1.rand(0..1) * 2 - 1))

    m = (prng1.rand(1..$constant_bound))
    n2 = (prng1.rand(1..$constant_bound) * (prng1.rand(0..1) * 2 - 1))
    n1 = (prng1.rand(1..$constant_bound) * (prng1.rand(0..1) * 2 - 1))
    p1 = (prng1.rand(1..$constant_bound) * (prng1.rand(0..1) * 2 - 1))
    p2 = (prng1.rand(1..$constant_bound) * (prng1.rand(0..1) * 2 - 1))

    a = m^n1
    b = m^n1
    c = n2
    d = p2
    e = n2^x + p2^y

    if b/a == d/c || a/b == c/d || (c == 0 && d == 0) || (a == 0 && b == 0)
        next
    end

    # Get intersection of two linear equations
    x_y = solve_matrix(a, b, c, d, e)
    # => accept the random set if intersection is within the circle of radius 3,
    # and the acute intersection degree is larger than pi/4
    r = Math.sqrt(x_y[0, 0].to_f**2 + x_y[1, 0].to_f**2).round(0)
    angle = (Math.atan(b/a) - Math.atan(c/d)).abs
    break if r < 3.0 && angle >= PI/4 && x_y[1, 0].abs >= 0.5 && x_y[0, 0].abs >= 0.5

    r = r + prng1.rand(4..8)
end
```

Figure 3.4 Code snippet that generates three constraints for each student. The parameters (radius, angle of intersection and the radius of the circle can be adjusted according to class size to ensure the variations between students.
3.2.2 Printing Canonical SOP / POS

In this assignment, each student is given a Boolean function $F$ and must write a program to print out the canonical SOP or POS (in minterms or maxterms) of function $F$. The function $F$ is defined by 3 or 4 variables plus $X$ and $Y$, where $X$ and $Y$ are user inputs. Plugging in the user-provided values of $X$ and $Y$ to function $F$ results in a Boolean function that depends only on the other variables. For example, for a Boolean function $F_{x,y}(A, B, C, D) = (A' + B + D + X')(B + C' + D' + Y)$, plugging $X = 0$ and $Y = 0$ into the function $F_{x,y}$ gives $F_{0,0}(A, B, C, D) = B + C' + D'$. In this way, the assignment is parametrized by $X$ and $Y$, as the output depends on the input of the function.

SOP stands for “sum of products” and POS stands for “product of sums”. Minterms are elements to construct SOP and maxterms are used to construct POS. Minterms are the logical AND of a set of variables and maxterms are the logical OR of a set of variables. The summation (logical OR) of minterms is equivalent to SOP and the product (logical AND) of maxterms is simply POS. Minterms can be written as "$m_x$" and maxterms have the form "$M_x$", where $x$ is an integer called term number which is used to determine the Boolean function represented by minterms or maxterms. The reason for introducing these representations to this assignment is the minterms and maxterm representations of a Boolean function is simple and clear, and does not require choices nor optimizations. Using the truth table of a Boolean function is convenient to obtain the SOP in minterms of the Boolean function. By identifying the true results in the truth table, one can obtain the SOP quickly by writing the position of the true results as term numbers. For example, if a Boolean function has true results at the 0th, 2nd, and 4th place in its truth table, the canonical SOP in minterms is simply $m_0 + m_2 + m_4$. For canonical POS, the only difference is to find the false results instead of true results in the truth table. To complete this assignment, students need to write loops to iterate through all of the truth assignments of the variables in the given Boolean function. Whenever the result of a truth assignment matches their expectation
(determined by printing SOP or POS), they need to print the term expression (m for minterms and M for maxterms) and term number, calculated with the bitwise operators, together using a “printf” statement.

One potential problem of this assignment is that the minterms or maxterms representation of a Boolean function can be found manually. To prevent students from doing the coding assignment manually, the assignment generator makes sure that the Boolean function given to a student has the opposite form of what the student is supposed to print.

The variations in this assignment includes the form to print (SOP or POS), the names and the number of variables in the given Boolean function, and the clauses of the Boolean function. All of the Boolean functions generated by the assignment generator have three clauses, and each clause contains 3 literals of 4 variables or 2 literals of 3 variables plus one literal of X or Y. The assignment generator first randomly selects three sets of variables out of all the possible combinations of 3 out of 4 or 2 out of 3 variables, then randomly appends X or Y to each set. The resulting three sets of variables can be used to generate the three clauses of the Boolean functions by randomly negating variables. Figure 3.5 is the code snippet of the assignment generator.

```python
30    # => description: this function randomly generates a sum statement
31    # => input: vals is a list of variables
32    # => return: the sum statement
33    def rand_entry_sop(vals):
34        flag = 0
35        statement = []
36        line = []
37        vals.each( |val, idx|
38            statement << (val.to_s + ( (flag = get_sign()) ? "" : "")
39                line << (( flag ? ~"" : "") + val.to_s)
40            )
41        statement_s = statement.join(" + ")
42        code_s = line.join("|")
43    return statement_s, code_s
44
```

Figure 3.5  Code snippet from the assignment generator. The input “vals” is a list of variable names. This function generates a Boolean function in SOP by randomly negating the variables in the input list. “Statement_s” is a human readable notation of the Boolean function, and “code_s” is the statement of the Boolean function is C.
3.2.3 Printing a K-map

Similar to the previous assignment, in this assignment, each student is given a Boolean function defined by 3 or 4 variables plus X and Y, where X and Y are user inputs. Plugging in X and Y values forms a new Boolean function. The difference between these two assignments is that instead of printing a canonical form of the given Boolean function, this assignment asks students to print the K-map of the Boolean function. The first two variables (in alphabetic order) are used to label the columns of the K-map, and the last one or two variables are used to name the rows. One significant feature of K-maps is that the bit patterns in adjacent columns or rows should have only one different bit, which is known as the Gray code. The major problem for students in this assignment is to print the Gray code. To complete this assignment, students should first complete the routine steps (implementing prompt “printf” and “scanf” statements). Afterward, they need to use loops to iterate through the truth assignments of variables and print out the content of the K-map. But using loops alone cannot produce a Gray code. There are many implementations for printing Gray codes; the gold code uses an XOR operator. Figure 3.6 is the code snippet that demonstrates how to print Gray codes with XOR operator.

```c
int n, t, w, bgc, result;
gold_printf("***************\n");
gold_printf("*     n t *\n");
gold_printf("* 00 01 11 10 *\n");
gold_printf("*        *\n");
for (w = 0; w < 2; w++)
{
    gold_printf("w=%d\",w);
    for (n = 0; n < 2; n++)
    {
        for (t = 0; t < 2; t++)
        {
            bgc = n ^ t; /* Gray code calculation */
            result = (-n|~w|~y)&(~bgc|~w|x)&(n|bgc|x);
            gold_printf("%3d", 1 & result);
        }
    }
gold_printf("\n");
}
return 0;
```

Figure 3.6 Code snippet that use XOR operator to generate Gray codes. This K-map has three variables (w, n, and t). Variables n, and t are variables for the columns.
This session focuses on the discussion of the design of skeleton code and modifications made by the assignment generator to complete the skeleton code. Due to the similarity between this and the previous assignment, many functions from the previous assignment generator can be directly used in this assignment. Since there are two possible values for the numbers of variables, a straightforward approach of designing the skeleton code is to have one skeleton code for each condition. In the skeleton code of the parametric coding assignments, the brackets “<” and “>” are the mark of modification. The string between these two symbols is the identifier of the position. The assignment generator substitutes the brackets and identifier strings in the file to corresponding elements. The strings to be substituted have two different types: the name of variables and valid C programming statements. Figure 3.7 is the code snippet from the skeleton code for printing three variable K-maps. All of the marks with the same identifier string should be substituted with exactly the same thing (variable name or statements). “<a>”

```c
int function_result(int <a>, int <b>, int <c>, int X, int Y)
{
    <b> = <a> ^ <b>; /* Gray code calculation */
    return (<statement>);
}

int main()
{
    int X, Y;
    printf("Please input X and Y values\n");
    scanf("%d %d", &X, &Y);
    int <a>, <b>, <c>, bgc;
    printf("" " ");
    printf("A\n");
    printf("" " ");
    printf("B\n");
    for (<c> = 0; <c> < 2; <c>++)
    {
        printf("<c>=%d","<c>);
        for (<a> = 0; <a> < 2; <a>++)
        {
            for (<b> = 0; <b> < 2; <b>++)
            {
            }

        }
        printf("%d", 1 & function_result(<a>,<b>,<c>,X,Y));
    }
    printf("\n");
    return 0;
}
```

Figure 3.7 Code snippet of the skeleton code for printing three variable K-maps.
in Figure 3.7 should be changed to the first (in alphabetic order) variable in the student’s Boolean function and all of the “<a>” strings in this skeleton code should be changed to the same variable. “<b>” is the name of the second variable and so on. The “<statement>” should be a valid C programming statement and in this example, it should be replaced by the Boolean function given to students.

3.2.4 Bit-sliced Logic Simulation

In this assignment, students must implement a function to simulate a bit-sliced N-bit unsigned number comparator. There are three numbers to read from the console: num1, num2, and N. The variables num1 and num2 are the two unsigned numbers to be compared, and N is the number of bits to compare (the comparator should compare the last N bits of num1 and num2). The bit-patterns used to represent three possible comparison results ("<", ">", and "=") are different for each student. The bit-patterns are simply 2-bit binary numbers. Notice that there are two ways to compare two numbers, starting from the least significant bit or from the most significant bit. This assignment specifies which direction a student must implement.

Students must use loops to simulate the behavior of a bit-sliced comparator. Recall that a bit-sliced comparator has a design described in Figure 3.8. To complete this assignment, students should first derive the Boolean functions of P and Q represented by A, B, C1, and C0 manually (base on the

![Diagram of bit-sliced comparator](image-url)

Figure 3.8 Design of a slice of the bit-sliced comparator. Inputs A and B are the bit from two numbers to be compared, and inputs C1 and C0 represent the comparison result of the previous logic. P and Q are the outputs of this slice of the bit-sliced comparator logic.)
comparing direction and bit-pattern). With the derived Boolean functions, students can simulate the comparison of the bit-sliced comparator by implementing the Boolean function of P and Q in the loops.

One potential problem of this assignment is that students are able to take a shortcut to get the answer by using comparison operator in C directly. To prevent this, this assignment requires students to print out the output bits from each slice of the bit-sliced logic.

The assignment generator first selects the direction of comparison (starting from the LSB or from the MSB) and then randomly shuffles a list of four integers (from 0 to 3). The first three integers are considered as the bit-patterns for three comparison relationships ("<", ">", and "="). With these elements, the student-specific text specification in txt form is generated. Figure 3.9 is the code snippet that determines the output bits for each truth assignment. The challenge for this assignment generator

```python
66    def cmp_intrprt_LSB(ab, c):
67          if c == "x"
68              return "x"
69          end
70        if ab == "eq"
71              return c
72          else
73              return ab
74          end
75        end
76        end
77
78    def cmp_intrprt_MSB(ab, c):
79          if c == "x"
80              return "x"
81          end
82        if c == "eq"
83              return ab
84          else
85              return c
86          end
87        end
88        end
```

Figure 3.9 Code snippet that determines the output bits of a bit-sliced logic. Here, “ab” is the comparison relationship between two input bits and “c” is the comparison relationship of the previous slice of the bit-sliced logic. “eq” stands for the relationship of equal and “x” means do not care since there is always one integer between 0 to 3 is not a valid bit-pattern.
is to obtain the Boolean functions for the output bits. Fortunately, the Boolean functions in the gold code do not need to be in the shortest form. The assignment generator first generates the whole truth table and then uses the truth table to obtain canonical SOP or canonical POS, whichever has fewest clauses. To compare from the least significant bit, the relationship between the input bits A and B overwrite the result of the previous logic unless A equals B. As for comparing from the most significant bit, the result of previous logic overwrites the relationship between current inputs unless the previous result is equal. In this assignment, the skeleton code also consists of two files: one for comparing from the LSB and the other for comparing from the MSB. Figure 3.10 is the snippet of skeleton code of this assignment. “<eq>” will be replaced with the integer of the bit-pattern of equal. “<statementz1>” and “<statementz0>” are the Boolean functions of output bits.

3.2.5 Finding the Stable States of a Sequential Feedback Circuit

In this exercise, students write a program that identifies all stable states for a sequential feedback circuit given a set of inputs to the circuit. The circuit consists of three cross-connected, two-input gates, and each student is given a randomly selected combination of gates (chosen from AND, OR, XOR, XNOR, NAND, and NOR). For the output, students must print all of the inputs (input1, input2, and input3) and the outputs of all of the gates for each stable state. To help students with this assignment, an example

```c
/* Initialize c1 and c0 to equal */
bit_mask = 1;
status = <eq>;
c1 = (status >> 1) & 1;
c0 = status & 1;

for(i = 0; i < NumOfBit; i++){
    ai = (num1 & bit_mask)>>i;
    bi = (num2 & bit_mask)>>i;
    z1 = <statementz1> ; /* Boolean function for P */
    z0 = <statementz0> ; /* Boolean function for Q */
}
```

Figure 3.10 Code snippet of skeleton gold code that compares two numbers from the least significant bit.

21
code that finds all steady states of an S-R latch is provided. To find steady states, the student’s function should assume a value for Q and calculate the resulting values of each gate. Finally, the function checks the stability of the logic by comparing the resulting Q with the assumed value of Q.

The topologies of the three gates are shown in Figure 3.11, and Figure 3.12 is the skeleton solution code. In general, this assignment is not challenging as long as the students are familiar with the idea of steady

![Figure 3.11](image)

**Figure 3.11** Three topologies of three cross-connected, two-input gates. All of these topologies have three user inputs and an output.

```c
for (assumed_val = 0; assumed_val <= 1; assumed_val++){
    Q_orig = <Q_orig>;
    temp = assumed_val;

    gate1_out = <gate1> ;
    gate2_out = <gate2> ;
    Q = <gate3> ;

    if ( (Q_orig & 1) == (Q & 1) ){
        Q_orig = Q_orig & 1;
        Q = Q & 1;
        printf("Stable state found at %d %d %d %d %d\n", in1,in2,in3,gate1_out&1,gate2_out&1,Q);
    } 
}
return 0;
```

**Figure 3.12** Code snippet of the skeleton gold code
state and using logic operators in C. In the skeleton code, “<Q_orig>” is the statement that calculates the assumed value of Q. For topology 1 and topology 3, “<Q_orig>” is simply “assumed_val”. Topology 2 is the special case in which the output Q is not part of the feedback loop and “<Q_orig>” should be replaced with the gate 3 logic with inputs: “assumed_val” and “in3”.

3.3 Result of Production Run in Semester Fall 2016

These parametric coding assignments were put into production run in ECE120 this semester (Fall 2016). These assignments were distributed weekly as optional exercises for students to practice their programming skills and to review course material. Assignments were distributed weekly on Friday starting in the third week, after the course introduces C programming. In this semester, ECE120 approximately had an enrollment of 400 students. Although the instructors of the course encouraged students to try these exercises, only 117 people signed up before the first parametric coding assignment was distributed and 23 more enrolled later. With around one-quarter of the total students signed up, the largest number of submissions of an assignment, the first assignment, is 25, which is also around one-quarter of the students who signed up for the exercise. This result is not unexpected since the students received no extra credit or other direct benefit by completing these exercises. However, the limited submission still provided valuable feedback to improve the assignments.

Since the first assignment (implementing multi-region function) had the most submissions, a careful analysis was conducted on this assignment. The latest submissions of all of the incomplete submissions revealed ambiguity in the assignment specifications. There are 13 out of 15 incomplete submissions that failed to pass all the test cases having incorrect “printf” or “scanf” formats. It is possible that the students stopped debugging their assignments because the results of their manual testing seem to be flawless and spending too much time on the optional exercise does not provide them with enough
benefits. However, this type of error is so common that we cannot ignore it. Table 1 is the detailed list of error and explanation (error position is marked in a different color) of incomplete submissions. In the table, student 4’s implementation is incorrect because divide operator used on integers in C introduces serious truncation error. However, given that the students in ECE120 are new to C programming, this error is understandable but the assignment specification should point out this problem.

Table 1 Statics of incomplete submissions

<table>
<thead>
<tr>
<th>Student</th>
<th>Latest error message</th>
<th>Incorrect student code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input 3 integers:&quot;);</td>
</tr>
<tr>
<td>2</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input four integers\n&quot;);</td>
</tr>
<tr>
<td>3</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input 4 integers: \n&quot;);</td>
</tr>
<tr>
<td>4</td>
<td>Output value of function F</td>
<td>if (y&gt; (-5/4)-((3*x)/4))</td>
</tr>
<tr>
<td>5</td>
<td>Prompt printf not found</td>
<td>Submission does not have any valid change.</td>
</tr>
<tr>
<td>6</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please enter 2 integers.\n&quot;);</td>
</tr>
<tr>
<td>7</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input 3 integers: &quot;);</td>
</tr>
<tr>
<td>8</td>
<td>Scanf not found</td>
<td>scanf(&quot;%(d,%d)&quot;,%x,&amp;y);</td>
</tr>
<tr>
<td>9</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input N integers.\n&quot;);</td>
</tr>
<tr>
<td>10</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please input 2 integer.\n&quot;);</td>
</tr>
<tr>
<td>11</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please enter values of x, y and j: \n&quot;);</td>
</tr>
<tr>
<td>12</td>
<td>Output printf’s format</td>
<td>printf(&quot;F(%d,%d)=%d&quot;,x,y,out);</td>
</tr>
<tr>
<td>13</td>
<td>Incorrect prompt printf’s format</td>
<td>printf(&quot;Please enter 4 integers.\n&quot;);</td>
</tr>
<tr>
<td>14</td>
<td>Incorrect output printf’s format</td>
<td>printf(&quot;F(%d,<em>%d,</em>%d,%d) = %d\n&quot;</td>
</tr>
<tr>
<td>15</td>
<td>Prompt printf’s format</td>
<td>printf(&quot;Enter three integers&quot;);</td>
</tr>
</tbody>
</table>
Another interesting assignment to look into is the third assignment (Print K-map). Because this assignment is similar to one question in the students’ required homework, this assignment had high percentage of completion: 9 out of 11 submissions completed it. One of the two incomplete submissions has an incorrect format in prompt “printf” statement, and the other failed because the student hard-coded the result. Figure 3.13 is the snippet of the hard-coded part. K-maps printed by this function definitely looks correct; however, the assignment required students to use two different “printf” statements to print the column bit-patterns and the entries of the K-map. Fortunately, this code did not pass all the tests. Unfortunately, the grading component failed to identify the hard-coded behavior.

```c
/* TO DO: Replace "AB" in the second printf with your own first two variables. */

printf("**********************\n");
printf("  PS \n");
printf("  00 01 11 10 \n");
printf("\n");
printf("Ww%d%d %d %d %d\n",0,0,f(0,0,0,0,X,Y),f(0,0,1,0,X,Y),f(0,1,0,0,X,Y),f(0,1,0,0,X,Y));
printf("Ww%d%d %d %d %d\n",0,1,f(0,0,0,1,X,Y),f(0,0,1,1,X,Y),f(0,1,0,1,X,Y),f(0,1,1,1,X,Y));
printf("Ww%d%d %d %d %d\n",1,0,f(0,1,0,0,X,Y),f(1,0,0,0,X,Y),f(1,0,0,1,X,Y),f(1,0,1,0,X,Y));
printf("Ww%d%d %d %d %d\n",1,1,f(0,1,0,1,X,Y),f(1,0,0,1,X,Y),f(1,0,1,1,X,Y),f(1,1,0,1,X,Y));

Figure 3.13 Code snippet of a student’s submission.
4. Conclusion

The assignment specification is the primary and the most important method for students to figure out the topic and requirement of an assignment, but students, most of the time, fail to read the specification slowly and carefully. Therefore, the specification needs to emphasize the significant content and the parts that can be easily ignored, for example, the format string of “printf” format. The reason that the parametric coding assignments restrict the “printf” format is because the grading of these assignments purely depends on the output of “printf” statements since ECE120 does not include material on data structures and arrays.

Errors made by inexperienced programmers often exceed the imagination of assignment designers, and thus it is difficult to design a grading component that can always provide precise feedback. However, the current grading component can definitely be improved by conducting intensive testing and having more students participating in these assignments. This need leads to another currently unsolved problem: how to increase the number of submissions without affecting student grades.
References


Appendix A: Feedback messages of each assignment

A.1 Error Conditions and Messages of Week3 (Multi-region function)

If the student code cannot be compiled, no test case will be generated.

The grading script will first check if the prompt printf exists. The following message will be generated if the code has no printf or the printf does not follow the requirements at all.

"Prompt printf was not detected.
You need to print a prompt before reading inputs from the keyboard.
Please make sure that your printfs follow the requirement specified in the PDF file."

If grading script is able to identify the prompt printf (if the printf prints out “Please …”), the grading script will then verify the format of this printf. If the format is incorrect, it generates:

"Prompt printf format specification is incorrect.
Please make sure that you print out your number of inputs.
Also check the newline character at the end."

Then, the grading script checks the number of variables in the prompt printf. Both “Please input 2 integers.” and “Please input %d integers.” are acceptable. If the number is incorrect, the script generates this message:

"Please use the correct number of integer variables (see mp_spec.txt) in your prompt printf’s format."
Check for the existence of scanf. Generate this error if it is not found.

"Scanf was not found. You need a scanf to read inputs from keyboard."

Check the number of format specifiers. If it does not match with the gold input numbers, generate this error.

"Scanf does not read the correct number of integers."

The grading script will then move on to check the output printf. The logic of checking output printf is similar to that checks the prompt printf. If there is no printf printing out “F( ...”, the following error message will be generated:

"Output printf was not detected.

You need to print all the inputs and the result of your function F.

Please use the format specified in the PDF file."

If grading script is able to identify the output printf but the format is incorrect, it generates the following message:

"Output printf has an incorrect format string.

Please check the spaces and newline character and make sure that you print out all the required variables.

Also, please do not hard code the variable values."

This error message is generated if students do not provide enough parameters for the output printf. However, it should never be generated because the format check makes sure that this would never
happen or the compiler would throw a warning if the number of parameters does not match with the number of format specifiers.

"In your output printf, please print out the values of all of the input variables."

If the student modified the console input during the calculation, this error message is generated.

"In the output printf, at least one of the input variables printed is different from its original value.
Please print out the original input values in the output printf."

Verify the calculation result after the format of the output printf is checked. If the result is different with the gold version, the grading script generates this error message with information of the test case that this student code failed.

"The output value of your function F is incorrect.
Please check the conditions and calculations of function F and the output variable.
Testcase: #{err_example}" 

Undetermined error: it is Bet’s fault.

"Undetermined error: #{line}
Please send an email to let Bei know about this!"

Execution takes longer than 5 mins.

"Execution out of time"

Message for correct code.
"You have passed all the tests!"

A.2 Error Conditions and Messages of Week4 (Printing Canonical SOP/POS)

If the student code cannot be compiled, no test case will be generated.

The grading script will first check if the prompt printf exists. The following message will be generated if the code has no printf or the printf does not follow the requirements at all.

"Prompt printf was not detected.
You need to print a prompt before reading inputs from the keyboard.
Please make sure that your printfs follow the requirements specified in the PDF file."

If grading script is able to identify the prompt printf (if the printf prints out “Please ...”), the grading script will then verify the format of this printf. If the format is incorrect, it generates:

"Prompt printf format specification is incorrect.
Please make sure that you print out your number of inputs.
Also check the newline character at the end."

Check for the existence of scanf. Generate this error if it is not found.

"Scanf was not found. You need one and only one scanf to read inputs from the keyboard."

Check the number of format specifiers. It does not match with the gold input numbers, generate this error.
"Scanf does not read the correct number of integers."

Once the prompt printf is verified, grading script begins to wait for the printf which prints the function name, variables’ names and operator. If the grading script cannot find it:

"You need to use a printf to print the name of the function, variables' names and the operation (Pi or Sigma)."

By comparing the format strings of the student code and that of the gold code, the grading script determines if the function’s name, variables’ names and the operator are correct. For differences that occur before the “=” in “F(X,Y,Z)=Pi(“:

"Please check your function name and variables.
And make sure that there are no spaces in the format string."

If a difference occurs right after “=” and before the last bracket, the problem is with the operator in the student string.

“The operator for your function is incorrect.”

Otherwise, the problem is because the student forgot the last bracket:

"Your printf which prints the function information has a little problem.
Please check the format again."

After the two printf statement above is verified, the grading code begins to look for the format string starts with maxterm or minterm specifier, "M" or “m”, the grading script will record the term number from argument list to an array for comparison. If the script cannot find it:
"Now please use a printf to print minterms or maxterms of your Boolean function."

If the student’s term type is different from the gold code’s:

"Please use correct term type: M for maxterm and m for minterm"

If student code uses difference kinds of expression of terms:

“Please use consistent term expression.”

If the terms are used correctly, then the script checks the term numbers:

“At least one of your terms is incorrect. Test case:

If the execution of student code is longer than 5 minutes:

"#{func_name} execution out of time"

If the student code does not generate any error:

"You have passed all the tests!"

A.3 Error Conditions and Messages of Week5 (Printing K-map)

If the student code cannot be compiled, no test case will be generated.

The grading script will first check if the prompt printf exists. The following message will be generated if the code has no printf or the printf does not follow the requirements at all.
"Prompt printf was not detected. You need to print a prompt before reading inputs from the keyboard. Please make sure that your printfs follow the requirements specified in the PDF file."

If grading script is able to identify the prompt printf (if the printf prints out “Please …”), the grading script will then verify the format of this printf. If the format is incorrect, it generates:

"Prompt printf format specification is incorrect. Please make sure that you print out your number of inputs. Also check the newline character at the end."

Check for the existence of scanf. Generate this error if it is not found.

"Scanf was not found. You need one and only one scanf to read inputs from the keyboard."

Check the number of format specifiers. It does not match with the gold input numbers, generate this error.

"Scanf does not read the correct number of integers."

Once the prompt printf is verified, grading script begins to wait for the printf which prints the name of variables at the columns of a Kmap. If the names are the difference between the student’s output and gold’s output:

"The variables at the Kmap header are not correct. Please replace the original "AB" with your first two variables."
After the header printf, the script expects a printf which prints the variables at rows. If no such printf is found:

"You need a printf to print out name and value of variable(s) at the most left column."

If the names of the variables of a student code are different with that of the gold code:

"The name of variable(s) at the left column of Kmap is(are) incorrect."

And for a difference between the value of variables:

"The value of the variables at the row is incorrect."

Count how many times the row variable printf was called (number of rows of the Kmap).

Compare the row number with that of the gold code.

"The number of rows of your Kmap is incorrect."

After the row printf is verified, the script begins to process the printf that prints the value of cells in the Kmap. If not such printf is detected:

"You need a printf to print the value of cells of your Kmap."

For a difference number of entries:

"The number of entries in K-map is incorrect."

Compare the values of cells. If there is any difference, print:

"At least one of the entries in your K-map is incorrect.

Test case: X and Y are #{err_example}"
If the execution of student code is longer than 5 minutes:

"#{func_name} execution out of time"

If the student code does not generate any error:

"You have passed all the tests!"

A.4 Error Conditions and Messages of Week6 (Simulating Bit-sliced Logic)

If the student code cannot be compiled, no test case will be generated.

The grading script will first check if the prompt printf exists. The following message will be generated if the code has no printf or the printf does not follow the requirements at all.

"Prompt printf was not detected.
You need to print a prompt before reading inputs from the keyboard.
Please make sure that your printfs follow the requirements specified in the PDF file."

If grading script is able to identify the prompt printf (if the printf prints out “Please ...”), the grading script will then verify the format of this printf. If the format is incorrect, it generates:

"Prompt printf format specification is incorrect.
Please make sure that you print out your number of inputs.
Also check the newline character at the end."

Check for the existence of scanf. Generate this error if it is not found.
"Scanf was not found. You need one and only one scanf to read inputs from the console."

Check the number of format specifiers. It does not match with the gold input numbers, generate this error.

"Scanf does not read the correct number of integers."

Once the prompt printf is verified, grading script begins to wait for the printf which prints the output of each bit-sliced logic. The format of result printf should contain only "\%d" and an arbitrary amount of whitespaces. The grading script will read one integer if the format string starts with "\%d":

"Result of bit-sliced logic was not detected.
You need a printf to print the outputs of all the bit-sliced logics.
Also, please make sure that you read inputs from the console using correct data type.
Test case: num1, num2 and number of bits are #{err_example}"

After the result printf is detected, the script compares the number of outputs:

"The number of results you should print is incorrect.
Test case: num1, num2 and number of bits are #{err_example}"

If the value of the first result is incorrect, it is probably because of uninitialized bit-pattern:

"The result of the first logic is incorrect.
Please check the initial condition of your comparator
Test case: num1, num2, and number of bits are #{err_example}"
And for a difference between the value of variables:

"The value of the variables at the row is incorrect."

Compare all of the results. If there is any difference between the student’s outputs and gold code’s outputs:

"At least one of the printed results is incorrect. 
Test case: num1, num2 and number of bits are #{err_example}"

If the execution of student code is longer than 5 minutes:

"#{func_name} execution out of time"

If the student code does not generate any error:

"You have passed all the tests!"

A.5 Error Conditions and Messages of Week7 (Finding the Stable States)

If the student code cannot be compiled, no test case will be generated.

The grading script will first check if the prompt printf exists. The following message will be generated if the code has no printf or the printf does not follow the requirements at all.

"Prompt printf was not detected.
You need to print a prompt before reading inputs from the keyboard."
Please make sure that your printfs follow the requirements specified in the PDF file.

If grading script is able to identify the prompt printf (if the printf prints out “Please ...”), the grading script will then verify the format of this printf. If the format is incorrect, it generates:

"Prompt printf format specification is incorrect.
Please make sure that you print out your number of inputs.
Also check the newline character at the end."

Check for the existence of scanf. Generate this error if it is not found.

"Scanf was not found. You need one and only one scanf to read inputs from the console."

Check the number of format specifiers. It does not match with the gold input numbers, generate this error.

"Scanf does not read the correct number of integers."

Once the prompt printf and scanf are verified, the script begins to look for the result printf. If the result printf is not found:

"Result printf was not caught. You need to print the inputs and internal states. You could also fail this test if your function cannot find any stable state."

If the format of result print is incorrect:

"The format of your result printf is incorrect. Please check the spaces and newline character in the format string."
If the gold code finds no steady state for a set of inputs and student code finds exactly one steady state:

"The stable state you found is not stable.
Testcase: input1, input2, and input3 are #{err_example}"

If the gold code finds no steady state for a set of inputs and student code finds two steady states:

"The stable states you found are not stable.
Testcase: input1, input2, and input3 are #{err_example}"

If the gold code finds exactly one steady state for a set of inputs and student code finds no steady state:

“Your function has a missing stable state.
Testcase: input1, input2, and input3 are #{err_example}”

If the gold code finds exactly one steady state for a set of inputs and student code finds two steady states:

“Your function has one extra stable state.
Testcase: input1, input2, and input3 are #{err_example}”

If the gold code finds two steady states for a set of inputs and student code finds no steady state:

“Your function has missing stable states.
If the gold code finds two steady states for a set of inputs and student code finds only one steady state:

“Your function has one missing stable state.

Testcase: input1, input2, and input3 are #{err_example}"

If the number of steady states from student code matches that from gold code, the grading script begins to compare the outputs. If a student failed to print all the required entries:

“Please print out all required values in your result printf.”

For any unmatched entry in student’s result output:

There is one error in the first result printf

“Some value(s) in your result printf for the first state is(are) incorrect.
If your function cannot find all stable states, it is the test you probably fail.

Testcase: input1, input2, and input3 are #{err_example}”

Or

“Some value(s) in your result printf for the second state is(are) incorrect.
If your function cannot find all stable states, it is the test you probably fail.

Testcase: input1, input2, and input3 are #{err_example}”

If the execution of student code is longer than 5 minutes:

"#{func_name} execution out of time"
If the student code does not generate any error:

"You have passed all the tests!"