The University of Melbourne
SWEN90006: Software Testing and Reliability

Tutorial 3 Solutions

Semester 2, 2017

Introduction

The purpose of this tutorial is for you to gain a deeper understanding of control- and data-flow techniques, and to compare these to input partitioning techniques.

Talking About Sets

There are still questions about how to characterise, or describe, input/output domains and what level of description and how many words are necessary.

You should aim to get used to characterising sets of values using proper set notation - we are after all, dealing with sets. The set notation does not need to be completely formal for this subject, but we will encourage and expect a mathematical discourse as the subjects goes on and in the exam.

To get the feeling for the level of discourse required read the tutorial and assignment answers on the web-page. Here are some examples of what we expect.

Example 1 Start by giving meaning to all of the (mathematical) variables that you intend to use and then use them to describe the input domain (i.e the set of all inputs that you will use for testing).

(i) Let string be the set of alpha-numeric strings, $\mathsf{str} \in \mathsf{string}$ be any individual alpha-numeric string, and length be a function that returns the length of a string. The input domain is then the set string and the input condition is that

$$\mathsf{for\ all\ str} \in \mathsf{string}\ \bullet 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Equivalently we could have written the input condition as:

$$\forall\ \mathsf{str} \bullet \mathsf{str} \in \mathsf{string} \rightarrow 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Both are acceptable.

(ii) Let int be the set of all 32-bit integers and string be the set of all strings of characters. The input domain is then the set of all pairs

$$\mathsf{Input} = \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{x} \in \mathsf{string\ and\ y} \in \mathsf{int}\}.$$

Working With the Program

In this tutorial we will focus on the procedure `bubble`, found in Figure 1. The `main` program serves to show you how we intend to use `bubble`. Of course, `bubble` implements a bubble sort. The source code can be downloaded from the tutorials page of the LMS.

```
#include <stdio.h>
#define SIZE 10

void bubble(int data[SIZE], int size, int (*compare)(int, int));

main()
{
  int data[SIZE] = {11, 4, 8, 22, 15, 7, 8, 19, 20, 1};
  int counter, order;

  int up(int, int);
  int down(int, int);

  printf("Enter 0 to sort in ascending order and 1 to sort in descending order: ");

  scanf("%d", &order);

  if (order == 0) {
    bubble(data, SIZE, up);
  }
  else {
    bubble(data, SIZE, down);
  }

  for (counter = 0; counter < SIZE; counter++) {
    printf("%4d", data[counter]);
  }

  printf("\n");
}

void bubble(int *data, int size, int (*compare)(int, int))
{
  int pass, count;
  void swap(int *, int *);

  for (pass = 0; pass < SIZE - 1; pass++) {
    for (count = 0; count < SIZE - 1; count++) {
      if ((*compare)(data[count], data[count + 1])) {
        swap(&data[count], &data[count + 1]);
      }
    }
  }
}

void swap(int *A, int *B)
{
  int temp;

  temp = *A;
  *A = *B;
  *B = temp;
}
```

Figure 1: An implementation of bubble sort

Make special note of the third parameter to `bubble`, which is a function parameter. This parameter allows us to create a flexible sorting algorithm, which can be used to sort in either ascending or descending order.

Both functions take two integers and return an integer (which is intended to be a boolean value). For this tutorial consider just

```
int up(int A, int B) {
  return A < B;
}
```

and

```
int down(int A, int B) {
  return B < A;
}
```

as the possible comparison functions. Note that the functions `up` and `down` appear in the `main` function.

Another interesting aspect of both `bubble` and `main` below is that they declare *nested functions*. The functions `up` and `down` are declared within `main` and so can refer to any of the variables in `main` prior to their declaration, for example, they can refer to the variable `count`.

Your Tasks

Task 1

First, make sure you understand the program (expected to be done **before** the tutorial).

Task 2

Determine the input domains and output domains for the functions `bubble`. What is the set of possible functions that can be passed into the `bubble` function? Is this what you expected?

Next, what are the input conditions, and (perhaps a little more challenging) what are the output conditions?

> **Answer** This question is quite challenging and not one that is typically covered by the standard text books.
>
> The first thing to realise is that the set of functions is quite large, includes `up` and `down` whose declarations have the right types:
>
> ```
> int up(int A, int B);
> int down(int A, int B);
> ```
>
> However, in addition to these, we also must include any C function that has compatible types, such as the following:
>
> ```
> int plus(int A, int B) {
>   return A+B;
> }
> ```
>
> and
>
> ```
> int times(int A, int B) {
>   return A * B;
> }
> ```

These arithmetic functions have no real meaning in the context of `bubble` sorting. The question becomes, which set of functions should `bubble` be tested on? Each actual parameter function changes the semantics of `bubble` and its control flow graph.

The unexpected aspect of the `bubble` function is the fact that the types of the function parameter allow us to pass integer valued functions that do not return boolean valued results.

We will assume that we *know* the entire set of functions that will be passed as actual parameters to the `bubble` function and that this set consists of the functions {`up, down`}. Thus, we only need to consider two possible semantics for `bubble` and two possible control flow graphs.

The input domain for `bubble` is then obtained by looking at the parameters of `bubble`: `int*` $\times$ `int` $\times$ {`up,down`}.

If we were to remove our assumption that `up` and `down` were the only functions that could be used as input, then the input domain would be: `int*` $\times$ `int` $\times$ `((int` $\times$ `int)` $\rightarrow$ `int)`[1].

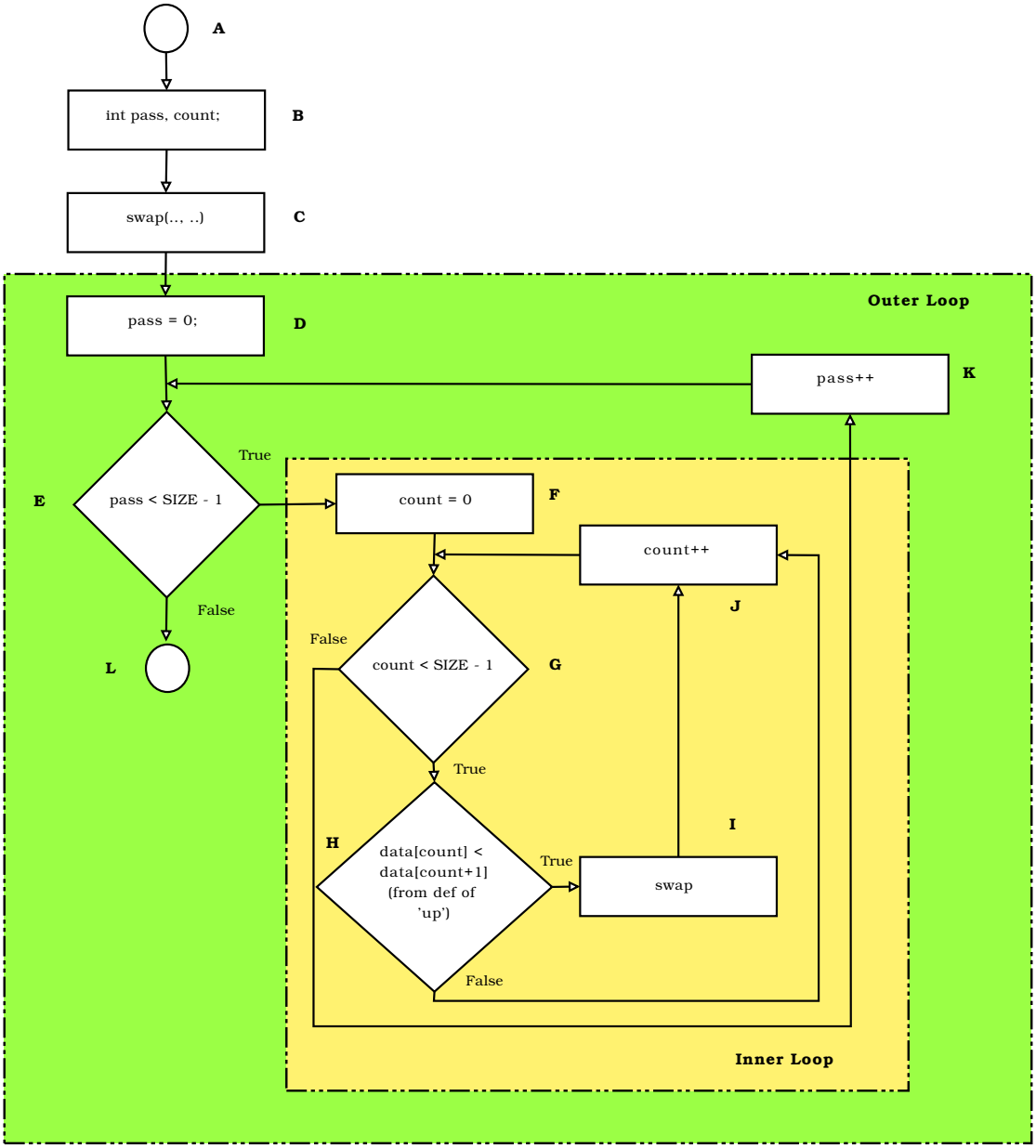
The input conditions are also straight-forward: `data` $\in$ `int*` and `size` $\in$ `int`. The third input condition is a little harder, but we do want `*compare` to behave like $\leq$ or $<$. We already have this for `up` and `down` and since we have assumed that we are limited to passing `up` and `down` to `bubble`, then this is all we require.

Task 3

Create a control-flow graph for the `bubble` function. Ideally, should you include the functions `up` and `down` in your control flow graph?

Answer

[1] This uses the syntax $A \rightarrow B$ to describe a function with input domain A and output domain B.



The functions `up` and `down` are not included in the CFG and do not affect the CFG.

Question 4

Do the branches in your control flow graph partition the input domain? Do the branches in your control flow graph partition the output domain?

Answer There are three branches in the control flow graph. Consider each one in turn.

(i) `pass < SIZE - 1` does not affect either the input domain or the output domain because `SIZE` is a constant. The value of the predicate `pass < SIZE - 1` is not dependent on any input variables.

(ii) `count < SIZE - 1` does not affect either the input domain or the output domain. The reason is the same as the above.

The reason that these branches do not partition the input domain is because of the fault in the program in which `SIZE` is used instead of `size`. This fault is difficult to detected from just reading the source code.

(iii) The choice of the `*compare` function and the third branch does create new subsets in the input domain, as it defines whether two elements should be swapped (less than or greater than). The choice of `*compare` function also affects the output domain. The choice of `up` partitions the output domain into arrays sorted into ascending order and arrays not sorted into ascending order. The choice of `down` partitions the output domain into into the set of arrays sorted into descending order and the set of arrays not sorted into descending order.

However, this program demonstrates a problem with control-flow analysis: if we use *just* control-flow analysis (e.g. if we had a program that performed control-flow analysis automatically to generate test inputs), then our test inputs may not relate to our specification particularly well. In the above case, we would generate only a handful equivalence classs due to the problem with the `SIZE` constant, and these are unlikely to achieve good coverage.

Task 5

Perform a static data-flow analysis on `bubble` for the variables `data` and `size`.

Answer

A simple way to attack data-flow problems is to write out a table with all the variables, and in which node of the control-flow graph each variables is defined, referenced, and undefined. In practice, we would create the entries in the table as we encountered each variable. We assign one of the states (u, d, r) to each variable. The notes for function parameters require us to tag `*data` as a P_{iw} parameter, and `*size` and `*compare` as P_i parameters.

The first three variables and the last are all part of the input domain, so they are defined at node A. For the others, they are still undefined at node A, so we explicitly list them as such in case the next use of the variable is a reference, leading to a ur-anomaly. Similarly, all variables except those in the output domain become undefined in node L.

Node	`data`	`size`	`compare`	`count`	`pass`	`swap`	`SIZE`
A	d	d	d	u	u	u	d
B							
C						d	
D					d		
E					r		r
F				d			
G				r			r
H	r		r	r			
I	dr			r		r	
J				d			
K					d		
L		u	u	u	u	u	

Now, let's look at the results. A quick scan down the column for `data` shows that the sequence **drdr** for this variable, which does not reveal any problems.

Next, we look at `size`. Looking down the column, we see that size is defined, and then undefined, without ever being referenced. This is a *d-u* anomaly. Where is the problem? The problem lies in the fact that we had intended to use `size` rather than `SIZE` in the body of the function. However, the program compiles because `bubble` is declared within the scope of `#define SIZE 10`.

Looking at other variables, we note that `count` and `pass` each contain a sequence **du** as the last two entries of its column. Referring back to the control-flow graph, we see that from node K, the flow goes through node E before arriving at node L. Node E references `pass`, so this is not a *d-u* anomaly. The lesson from this is that we must be careful when using tables to identify anomalies, because the control-flow of the program is discarded. We have to either return to the algorithm, or add the control-flow into the table.

Task 6

Design a set of black box test cases using whatever techniques you feel appropriate.

Question 7

What extra information does your data-flow analysis give you that your black-box test cases do not?

Answer

The important part of this question is to note the *d-u* anomaly from above. This anomaly cannot be revealed by testing the `main`+`bubble` program, because the parameter `size` is always passed the argument `SIZE`, so the behaviour is equivalent. If the `bubble` program is tested in isolation, it can be detected. The advantage of a static data-flow analysis is that this anomaly will *always* be detected, whereas using testing, it requires certain cases to be selected. If these are not selected, it may be missed.