The University of Melbourne
SWEN90006: Software Testing and Reliability

Tutorial 6 Solutions

Semester 2, 2017

Introduction

The aim of this tutorial is for you to familiarise yourself with the test oracles and random testing. This tutorial will help to gain an understanding of how to specify a test oracle to decide whether an arbitrary test case is correct, incorrect or undecidable, as well as how to randomly generate test cases that fit an operational profile.

Your Tasks

Repeat these tasks for both programs.

Task 1

Standard analysis: Determine the input/output domains and the input/output conditions.

Task 2

Use the specification, domains and conditions to determine if an arbitrary test input is correct, incorrect or otherwise undecidable.

> **Discussion** This is actually one of the key problems in reliability assessment – we need a means for detecting if the system has failed or not when a random test input is executed. Of course, being random we do not know in advance what the input will actually be and so have no way of anticipating the actual expected output. Instead we need to write an *oracle* that will return YES/NO answers.
>
> Of course the answer requires some thought:
>
> (i) The first thing is to understand exactly what is a failure for the program and the ways in which the program can fail.
>
> (ii) The second is to define an algorithm for detecting each kind of failure and an algorithm for detecting correct results (or non-failures).
>
> Typically, to detect failures we need to have some constraints on the input telling us what is valid and should return a correct result and what is invalid and should return an error or other error condition. Such constraints are often derived (yes! This is a creative process) from the specification or from the input/output conditions.

Task 3

Determine an automated means for generating random test cases by selection points from the input set.

Working With the Programs

We shall start this tutorial by using a rather simple example to illustrate the concepts, techniques and issues faced when working with oracles. This is followed by applying these techniques to the root finding program from tutorial 4 to gain some practice with practical examples.

First Program

This *toLower* is an application which takes a string on standard input and puts a corresponding string on standard output, with all upper case letters changed to the matching lower case letters (i.e. 'A' to 'a', 'B' to 'b', etc.). Strings may consist of all ASCII characters between 32 and 126 inclusive; characters outside this range are control characters and will cause an error message.

toLower Program

```
#include <stdio.h>
#include <string.h>

#define MAX_STRING 80

int main(int argc, char* argv[])
{
  char string[MAX_STRING];
  int i;
  while (fgets(string, MAX_STRING, stdin)) {
    for (i = 0; i < strlen(string) - 1; i++) {
      if (string[i] < 32 || string[i] == 127) {
        printf("Illegal character found.\n");
        return 1;
      }
      /* The upper case letters have ASCII codes
       * between 65 and 90 (90 to 65 = 26 letters)*/
      if (string[i] >= 65 && string[i] <= 90) {
        string[i] = string[i] + 'a' - 'A';
      }
    }
    fputs(string, stdout);
  }
  return 0;
}
```

Answer

(i) **Task 1:** We view the program as a function, or a relation between inputs and outputs thus:

$$\mathsf{toLower} \ : \ \mathsf{Input\ Domain} \rightarrow \mathsf{Output\ Domain}.$$

The specification gives us some of the information, but we also use the code itself to determine this. The program ignores the command-line arguments and uses fgets() to read in a string of at most 80 characters from the input. The string is 'NULL' terminated.

Consequently, the input domain is the set of all strings:

$$\mathsf{Input} = \mathsf{char}[]$$

or the set of strings that contain 80 characters or less, because call to `fgets` restricts the size to 80:

$$\mathsf{Input}_{80} = \{\mathsf{s} : \mathsf{char}[] \mid \#\mathsf{s} \leq 80\}\}$$

The output domain for the program is the same.

Why would we choose to test all strings rather than just strings of length less than or equal to 80? Put another way, why might we want to use Input instead of Input_{80}? The reason is that we would want to test strings of arbitrary length to ensure that $\mathsf{toLower}$ handled strings of different lengths properly – see the input conditions below – and this gives us a better testing range.

The input condition is that for any $\mathsf{s} \in \mathsf{Input}$ and for any $0 \leq i < 80$

$$\mathsf{ascii}(\mathsf{s}[\mathsf{i}]) \geq 32 \wedge \mathsf{s}[\mathsf{i}] \neq \mathsf{NULL}.$$

The output condition is that for any $\mathsf{s} \in \mathsf{Output}$ and for any $0 \leq i < 80$

$$32 \leq \mathsf{ascii}(\mathsf{s}[\mathsf{i}]) \leq 64 \ \vee \ 91 \leq \mathsf{ascii}(\mathsf{s}[\mathsf{i}]) \leq 126$$

(ii) **Task 2:** The way to design an oracle is to begin by looking at the ways in which the program can fail. To understand how the program can fail you need to examine the specifications for the program, the input conditions and the output conditions. We also use the failure taxonomy from the lecture notes.

Here are some ideas:

1. It can return an error on a valid input;
2. It can return a valid output on an invalid input;
3. It can return a string with an upper case character in the output thus violating the output condition; or
4. It can insert upper cases characters instead of removing them, thus violating the output condition.
5. It can return a string that is all lower case by removing the upper case characters instead of substituting them.
6. It can substitute the wrong lower-case character for the upper-case character.

Now, the next step is to think about how we can detect each of the failures listed above. That is, how can we implement this?

One way is to check that each ascii character in the output is between 32 and 64, or between 91 and 126, as specified by the output condition. But this is not complete, as it does not check point 5 above, for example. To check point 5, we could check that the length of the input and the length of the output are equivalent.

Checking points 2 and 6 is also difficult. Check point 6 can be checked by iterating over the list and checking that upper case character in the input is 32 ascii characters from the character in the output. Point 2 can be checked by iterating over the characters and seeing if any fall outside of the legal character set.

However, we can see that the oracle is already becoming as complex as the program itself. One thing we should aim for with a test oracle is simplicity: we want the test oracle to be correct, so making it as complex as the program itself means that it will not serve well as an oracle.

Instead, the best we can do is implement a *partial* oracle: one that checks only some of the output properties, or works on only a subset of the input domain. In this case, we choose to check that the output is in the output domain, and that the length of the

input and output are the same (checking point 5 above), and that all input characters are valid.

Let Input be the set from the input condition. Then we have the following:

```
if Test ∈ Input then
   return toLower(Test) ∈ Output ∧ #Test = #toLower(Test)
else
   if string[i] ∉ Input for some i then
      toLower(Test) returns "Illegal character found"
```

Can we do better? How many different kinds of failures from the list above does this oracle detect? Does it distinguish between them?

(iii) **Task 3:** For this program, we have not been given an operational profile, so we instead simply generate random strings. To do this, we would randomly select parameters based on:

- The length of a string; and
- The characters in the string.

So, randomly select the length of a string, and then, for each character in the string, randomly generate a value between 0 and 127 (the set of ASCII characters). This number can be type-cast to characters.

However, one would expect that users would mostly pass input to the program that will not generate an error. We could estimate the operational profile of this. For example, consider that users would generate a string of less than 80 characters approximately 95% of the time, and would generate strings with valid characters 90% of the time. Our operational profile would consist of:

Input	**Probability**
$\#\texttt{string} \leq 80$	95%
$\#\texttt{string} > 80$	5%
$\forall i \bullet \texttt{ascii(string[i])} \geq 32 \wedge \texttt{string[i]} \neq \text{NULL}$	90%
$\exists i \bullet \texttt{ascii(string[i])} < 32 \vee \texttt{string[i]} = \text{NULL}$	10%

So, we would randomly generate a string of 80 characters or less 95% of the time, and more than 80 characters 10% of the time. Given this length, randomly generate a valid string, and then 10% of the time, insert an invalid character into the string. We may even wish to break the operational profile into cases based on how many invalid characters are in the string.

Second Program

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $Lower$ is negative, and $Upper$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time).

Root Finding Program: A Simple Interval Bisection Method for Finding Roots

```
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535
```

```
double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */
    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */
    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as Lower or Upper is easy:
       we know that Lower is negative and we know that Upper is positive.
       Multiple fr by Lower and if the result is positive then fr must be
       negative! If the result is negative then fr must be positive! */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

  } /* end while */

  return xr;
}
```

Answer

(i) **Task 1:** We've seen this example before in tutorial 4 so we already know its input and output domains. In case you've forgotten, here they are again.

The function Bisection uses no other inputs other than those coming from the parameters so we can describe the program in terms of a function $\mathsf{Bisection} : \mathsf{Input} \to \mathsf{Output}$ as:

$$\mathsf{Bisection} : \mathsf{double} \times \mathsf{double} \times \mathsf{double} \times \mathsf{int} \to \mathsf{double}$$

The input domain is consequently $double \times double \times double \times int$ if we ignore the function parameter.

(ii) The next question is "*How can the Bisection function fail*"? Here are some ideas:

- If there is a root between Upper and Lower and Bisection returns a value that is not within error of the actual root.
- Otherwise, if there is no root between Upper or Lower the algorithm should return the nearest value within the interval: Upper if the root of the function is greater than Upper and Lower otherwise.
  A second type of failure occurs if the value returned by Bisection returns a value that is not within error of Lower if the actual root is below the interval and we have not reached iMax iterations. Alternatively, Bisection returns a value that is not within error of Lower if the actual root is above the interval (Case (C) above) and we have not reached iMax iterations.

So we need to write an oracle with the that will detect failures in the cases above. This may seem non-trivial at first glance, because we need to know the root of any arbitrary function that is passed to the program. However, we can write a passive oracle to check this. This is straightforward. For example, consider if the function passed was the following:

```
double f(double x) {
    return (x^2 - 4);
}
```

We are given a result from the program. To check whether this value falls within the error distance, we need only to write the following:

```
double result = Bisection(lower, upper, error, iMax, f);

if (-error >= f(result) >= error)
    return SUCCESS;
else
    return FAILURE;
```

So, we check the result against the function itself. If `f(result)` is equivalent to 0, or out by less than the error margin, then this will pass.

However, this does not consider the cases in which the program reaches the maximum number iterations and exits before finding the root of the function. It is problems such as these that make automated test oracles difficult to derive. This problem is particularly difficult because our specification is incomplete! It does not say what the output should be in this case. Can we improve our test oracle to work around this problem if we assume what the behaviour should be?

(iii) **Task 3:** Generating test cases for this program is certainly non-trivial. The largest problem is generating random functions to pass to the program. The most sensible way to get around this is to pre-define a set of functions that can be used, and randomly select from this set.

The second problem is knowing which values to choose for `Lower` and `Upper`. That is, we want to choose values such that the root falls between them; or at least, we would expect that, as part of the operational profile, this is what most users would want. If we know which functions will be passed to the `bisection` program in advance (which we have assumed above) then this becomes less of a problem, because we can calculate the root of the function (or even an estimate of the root), and select values of `Lower` and `Upper` that are either side of these.

Given an estimate of the root of the function that is being passed, we can randomly derive tests in the following way (assuming that the variable `root` represents the estimate):

(i) `Lower` to be uniformly distributed between `-sqrt(MAX_INT) - 2` and `root`, and `Upper` between `Lower` and `sqrt(MAX_INT)`.

(ii) `error` to be uniformly distributed between 0% and 20% of the interval `Upper - Lower`.
(iii) `iMax` to be uniformly distributed over $[0, MAX\_INT]$.

Putting the oracle and driver together, the main block of this would look something like:

```
Lower = Random() * (sqrt(MAX_INT) - 2.0) + root;
Upper = Random() * (sqrt(MAX_INT) - Lower - 1.0) + Lower;
error = Random() * 0.2 * (Upper - Lower);
iMax = Random() * MAX_INT;

double result = Bisection(Lower, Upper, error, iMax, f);

if (-error >= f(result) >= error)
    return SUCCESS;
else
    return FAILURE;
```