Assignment 4: Testing

1DT056: Programming Embedded Systems
Uppsala University

February 11th, 2011

Starting with this assignment, we will give points when assessing the solutions of the questions. A total number of 12 out of 20 points has to be achieved to pass the assignment. As before, the assignment as a whole will only be graded pass/fail.

Exercise 1 Unit testing a stateful module

Stateful modules are modules with internal state, which means that the behaviour of such components can depend on earlier interactions with the module. Testing a stateful module requires particular care, since the module first has to be brought in the right state for testing, and has to be reset after executing a test case and put back into its initial state. As an additional complication, after executing the test it is not easily possible to access the pre-state of the module anymore.

This means that an executable unit test case for stateful modules has the following general structure:

```
int testCase(void) {
  // set up module to be tested; prepare inputs (1)

  // call functions of module                   (2)

  // assess results of testing                  (3)

  // reset module                               (4)
  return testResult;
}
```

As an example, we consider the binary search tree datastructure given in question 3 of assignment 2 (`http://www.it.uu.se/edu/course/homepage/pins/vt11/search_tree_original.c`). This module is stateful, since the search tree is stored in form of global variables, and a side effect (in fact, the main effect) of the function "insertNode" is to modify the search tree.

1. First derive a general test oracle that checks whether the search tree is well-formed. A search tree is well-formed if, for any node n in the tree, the nodes reachable through $n.left$ carry data strictly smaller than $n.data$, and the nodes reachable through $n.right$ carry data strictly larger than $n.data$. Implement this oracle as a Boolean function **(2p)**

```
int treeIsWellformed(void) { ... }
```

2. Implement a function **(1p)**

```
int contains(int d) { ... }
```

that checks whether the search tree currently contains the datum d.

3. Write one or multiple executable test cases that achieve full decision coverage for the function "insertNode", considering the decisions **(4p)**

   *currentPtr,
   n->data < currentNode->data,
   n->data > currentNode->data.

   We assume that the search tree is initially empty, i.e., root is NULL. At point (1) in the test case, the search tree can be filled with some initial amount of data. At (2), one or multiple invocations of "insertNode" are performed. At (3), the result of (2) is checked by invoking the function "treeIsWellFormed", and by checking that all data inserted at (2) is in fact present in the tree (using the function "contains"). At (4), the tree is reset and emptied by removing all nodes from it.
   Justify why your test suite achieves decision coverage.

Exercise 2 MC/DC

In this question, we work with a simple implementation of binary search on integer arrays (the code as also available on the web page: `http://www.it.uu.se/edu/course/homepage/pins/vt11/binary_search.c`):

```
/* Pre-condition: ar is sorted,
                  arLength is the length of ar */
int binSearch(int *ar, int arLength, int el) {
  int lo = 0, hi = arLength;

  while ( lo + 1 < hi ) {
    int mid = (lo + hi) / 2;
    if ( ar[mid] < el ) lo = mid + 1;
    else if ( ar[mid] > el ) hi = mid;
    else return mid;
  }
```

```
  if ( lo == hi || ar[lo] != el ) return -1;
  return lo;
}
```

Given an array "ar" of integers (sorted in ascending order), the number "arLength" of elements in the array, and an integer "el", the function either determines an array index at which "el" occurs in the array, or it returns -1 in case there is no such index.

1. To be able to implement executable test cases for "binSearch", write a test oracle (a piece of C code) that assesses whether the result of "binSearch" is correct, according to the informal description given above. The test oracle should be formulated in terms of the variables "ar", "arLength", "el", and "result" (where "result" is the value returned by "binSearch"). **(2p)**
2. Find a minimal number of test inputs for "binSearch" (triples of sorted arrays "ar" and the inputs "arLength", "el") that achieve MC/DC coverage, considering the decisions **(4p)**

   $$\text{lo} + 1 < \text{hi}, \quad \text{ar}[\text{mid}] < \text{el}, \quad \text{ar}[\text{mid}] > \text{el},$$
   $$\text{lo} == \text{hi} \;||\; \text{ar}[\text{lo}] \;!= \text{el}$$

   If any of the conditions accesses "ar" outside of its bounds (or would do so in case it were executed, e.g., in case mid >= arLength), the condition can be considered to evaluate to false.
   Justify why your test cases achieve coverage, and why you have found a minimal number of test cases.

Exercise 3 Control-flow coverage for reactive program

Consider the program given in the solution for question 1 of assignment 2 (`http://www.it.uu.se/edu/course/homepage/pins/vt11/solutions2.pdf`). The control-flow graph of the function "switchTask" is given in Fig. 1.

1. Give one feasible and one infeasible execution path for this control-flow graph. **(1p)**
2. We test the function "switchTask" by simulating the whole program, using the test suite **(3p)**

$$TS = \{\text{testCase1}, \text{testCase2}\} .$$

The two test cases of the test suite are implemented as debug functions that provide appropriate stimuli to the program:

```
SIGNAL void testCase1(void) {
  swatch(0.5);                          // start up

  PORTC ^= 1;

  swatch(0.02);
  expect(0, 0.16); swatch(0.16);        // OFF
  swatch(0.04);
  expect(2, 1.0); swatch(1.0);          // ON 1

  PORTC ^= 1;

  swatch(0.02);
  expect(0, 0.5); swatch(0.5);          // OFF

  printf("Test succeeded!\n");
}
```

```
SIGNAL void testCase2(void) {
  swatch(0.5);                          // start up

  PORTC ^= 1;

  swatch(0.02);
  expect(0, 0.16); swatch(0.16);        // OFF
  swatch(0.04);
  expect(2, 1.76); swatch(1.76);        // ON 1
  swatch(0.04);
  expect(4, 0.98); swatch(0.98);        // ON 2

  PORTC ^= 1;

  swatch(0.02);
  expect(0, 0.5); swatch(0.5);          // OFF
```

```
    printf("Test succeeded!\n");
}
```

(the function "expect" is defined as in the solution for assignment 2). Note that the test cases always use "expect" and "swatch" in combination, since a call to the signal function "expect" returns immediately; "swatch" is used to suspend the execution of the main script until the "expect" function has finished its work.
Determine the statement and branch coverage of the function "switchTask" that is achieved by "testCase1" and "testCase2" individually. For branch coverage, it is not necessary to count the transition into the initial node 1.

3. Determine the decision and condition coverage achieved by testCase1 and testCase2 individually, considering the following decisions and conditions occurring in the "switchTask": **(3p)**

   GPIO_ReadInputDataBit(GPIOC, SwitchPin),
   count != -1,
   count == 10,
   count == 100.

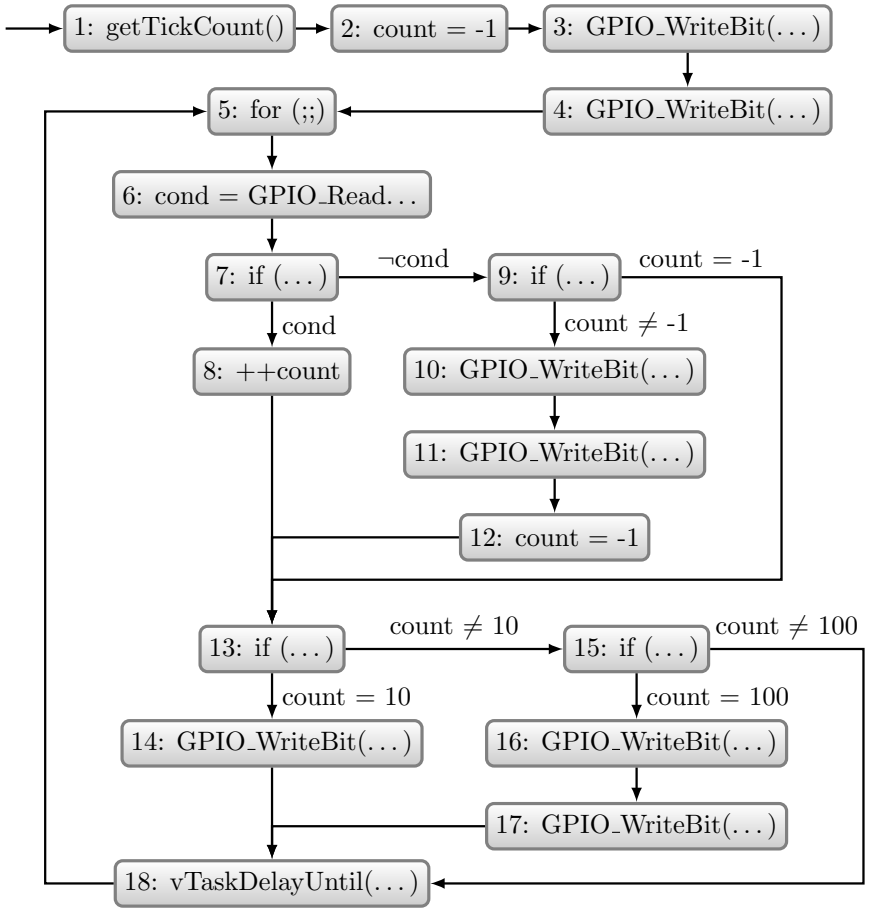

Figure 1: CFG of function "switchTask"

Submission

Solutions to this assignment are to be submitted by

Friday, February 18th, 2011.
You can submit your solution during the lab session
(8:15 – 10:00, room 1313) or by email to othmane.rezine@it.uu.se.

Make sure that you have specified your name and
your personnummer on your solution.

If you submit your solutions via email,
submit in form of a *single* PDF file!