HOMEWORK SOFTWARE TESTING

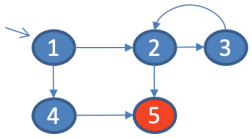
v2024_1

SET 1

(A&O CH. 2.1 – 2.3 + SLIDES OVER UNIT TESTING → FORMAL SPECIFICATION)

Note: A&O 2nd Edition: CH. 7.1 – 7.2.

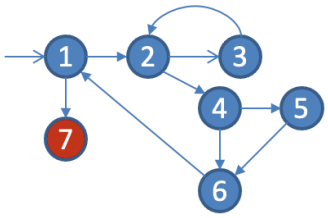
1. Recall first that we defined a **test path** over a control flow graph (CFG) as a path that abstractly represents the execution of some test case on the program represented by the CFG. It starts in the CFG's entry node, and in our setup we insist that it ends in one of its exit nodes.
   A **test set** is just a set of "test-cases". In terms of CFG, a test set is thus a set of test paths over a CFG.
   Consider the CFG below:



Node 1 is the entry node, 5 is the only exit node.

   Answers these questions:
   a. Is it possible to have a test set that covers all edges in this CFG, but not all its nodes?
   b. Is it possible for this CFG to have a test set that covers all nodes, but not all edges?
   c. Is it possible to have a test set (for this CFG) that covers all edges, but not all pairs of edges? By a 'pair of edge' it is meant here a pair of *consecutive* edges.
   d. What is the *minimum* size of a test set that would give you full coverage over pairs of edges of this CFG?

2. Consider the CFG defined in the question no 5 of Section 2.2.1 from A&O (2nd Ed: no 5 Sec 7.2.2):



1 is the entry node, 7 is the only exit node.

   Let's call the graph G, and let and nodes(G) be the set of all its nodes, and paths(G) be the set of all valid paths in G. Answers these questions:

   a. Consider the following two test requirements:

      TR1 : { [x] | x ∈ nodes(G) }
      TR2 : { s | s ∈ paths(G) and length(s) ≤ 1 } where length(s) refers to the length of a path s, which is defined as the number of edges in s (see the definition in A&O Section 2.1 (2nd Ed. Sec. 7.1)).

      Are these two requirements equivalent?

   b. How many prime paths start from node 1 ? (it is more than one)

c. How many prime paths pass node 3 ? (it is more than two) A path p is said to pass a node x if x is an element of p.
d. How many prime paths do we have in total?

3. Consider the program **MIN** with the following header:

int **MIN**(int *x*, int *y*)

The program returns the minimum of these two integers. Write a formal specification in terms of pre- and post-condition for this program. For this exercise you are not allowed to use conditional expressions (if-then-else expressions).
Use "retval" or "return" in your post-condition to refer to the program's return value. An example is shown below, showing a program that returns 1/x when given a non-zero x.

{* $x \neq 0$ *}
double **P**(int *x*)
{* retval = $1/x$ *}

The table below shows some of the key notations for writing pre/post-conditions.

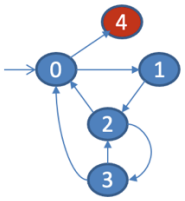
Use "**retval**" or "**return**" to refer to the return value of a method, e.g. as in retval>0.
conjunction: $p \wedge q$
disjunction: $p \vee q$
Implication: $p \Rightarrow q$
($\exists\ k: 0 \leq k < \#a : a[k]=0$) or (**exists** $k: 0 \leq k < \#a : a[k]=0$)
($\forall\ k: 0 \leq k < \#a : a[k]=0$) or (**forall** $k: 0 \leq k < \#a : a[k]=0$)

4. Consider the program **isMIN** with the following header:

boolean **isMIN**(int *m*, int *x*, int *y*)

The program checks if m is the minimum of *x* and *y*. If so, it returns true, and else false. Write a formal specification in terms of pre- and post-condition for this program. Use "retval" or "return" in your post-condition to refer to the program's return value. For this exercise you are not allowed to conditional expressions nor to reuse the MIN function from No. 3.

5. Consider the CFG below, of a program with nested loops:



0 is the entry node, and as usual red nodes are exit nodes (we only have one here).

Consider the prime path σ = [0,1,2,0] (the simple cycle that starts and ends in 0), and the following three test paths. How do the test paths cover the prime path σ? Is it by: *direct tour*, or by *side trip*, or by *detour*?

a. the test path [0,1,2,0,1,2,0,4] and the target path σ as defined above.
b. the test path [0,1,2,3,2,0,4] and the target path σ.
c. the test path [0,1,2,3,0,4] and the target path σ.

SET 2

(A&O CH. 4 + SLIDES OVER UNIT TESTING → FORMAL SPECIFICATION)

Note: A&O 2nd Ed: Ch. 6.

1. Imagine a program **Tax**(*source1,source2,source3*) that takes three parameters representing three different types of income that a person can have. You can imagine that *source1* represents regular income, *source2* represents benefit that a person may have, and *source3* represents income from a foreign source. The program calculates the amount of tax that the person owes to (or gets from) the state.

   Consider the following proposal partitioning of these inputs. They are "partitions", so they are meant to be non-intersecting.

		partitions			
		B1	B2	B3	B4
parameters	*source1*	zero	small value	normal value	invalid value
	source 2	zero	small value	normal value	invalid value
	source 3	zero	small value	normal value	invalid value

   In this example, each parameter is partitioned into four partitions/blocks named B1...B4. As the used concrete test values for each block we use the following:

		partitions			
		B1	B2	B3	B4
parameters	*source1*	0	5	2000	-1
	source 2	0	5	2000	-1
	source 3	0	5	2000	-1

   An abstract test for the program **Tax** can be expressed in terms of a tuple of the above partitions/blocks, e.g. (B1,B1.B1) or (B1,B2,B4). A concrete test is a tuple of concrete values, e.g. (0,0,0).

   a. Consider the following set of concrete tests (0,5,2000), (-1,-1,-1). Does this set give us full ECC?
   b. Consider the following set of concrete tests (0,5,2000), (5,2000,0), (2000,0,5), (-1,-1,-1). Does this set give us full ECC?
   c. Consider the following set of abstract tests (notice the patterns over the columns):

      (B1,B1 ,B1) (B1,B2,B4)
      (B2,B1,B2) (B2,B2,B1)
      (B3,B1,B3) (B3,B2,B2)
      (B4,B1,B4) (B4,B2,B3)

      Complete it to form a minimal test set that delivers full PWC. Please send me only the missing tests, sorted lexicographically.
   d. What is the minimum number of tests that would deliver full PWC for the program **Tax**?

2. Consider again the above partitioning of **Tax**. This time we want to apply MBCC, using the following choice of base blocks:

   {B1,B2} for *source1*
   {B1} for *source2*
   {B1} for *source3*

   Questions:

   a. Suppose *T* is a **minimum** test set that gives full MBCC with the above choice of base blocks. Does *T* cover all blocks of each characteristic?
   b. Is it required that *T* covers the combination (B3,B3,B3) ?
   c. Does *T* also give full PWC?
   d. What is the size of *T* in the above example?

3. Consider the program **TRIANGLE** with the following header:

   boolean **TRIANGLE**(int *x*, int *y*, int *z*)

   The program takes three **positive** integers representing the sides of a triangle. It returns true if these sides really form a triangle. This is the case if none of the side is equal to or larger than the sum of the other two. Else, the program returns false

   Write a formal specification in terms of pre- and post-condition for this program. Use "retval" or "return" in your post-condition to refer to the program's return value.

4. Consider this program:

   int **getMIN**(int[ ] *a*)

   The program takes a non-null and non-empty integer array *a*, and returns the smallest element of the array a. Write a formal specification in terms of pre- and post-condition for this program. Use "retval" or "return" in your post-condition to refer to the program's return value.
   We will need to use quantifiers to specify this program. There are two standard quantifiers you can use: **forall** (∀) and **exists** (∃).Few examples:

   - To say that for all valid indices *i* of the array *a*, *a*[*i*] is 0, we write: (∀*i*: 0≤*i*<#*a*: *a*[*i*]=0). Or alternatively, (∀*i*:: 0≤*i*<#*a* ⇒ a[*i*]=0).
   - (∃*i*: 0≤*i*<#*a*: *a*[*i*]=0) means that there exists one valid index *i* of the array *a*, such that *a*[*i*] is 0. We can also write it as (∃*i*:: 0≤*i*<#*a* /\ *a*[*i*]=0).

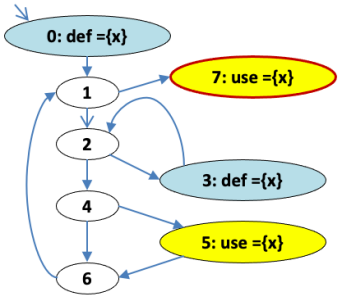
SET 3

(A&O CH. 2.2.2, 2.4.2, 7.1 + SLIDES OVER FORMAL SPECIFICATION)

Note: A&O 2nd Ed: Ch 7.2.3, 7.4.2. There is unfortunately no replacement for 7.1 in the 2nd Ed. So you need to somehow get them.

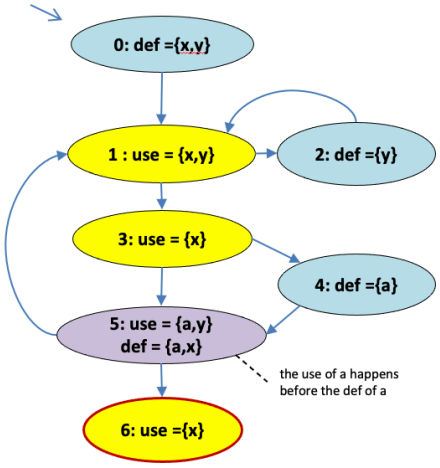
1. Some theoretical questions:
    a. Is it possible that a program contains a **prime path** which is not a **du-path** of any variable?
    b. Suppose a program P contains a variable x. Is it possible that it contains a du-path for x which is not a simple path?
    c. Suppose $[n_1,n_2,n_3]$ is a du-path for x. Where do writes to x take place?

2. Consider the following CFG, which has been decorated with the def/use information with respect to the variable x. The original program uses more variables, but we abstract those away in this problem.



    a. How many du-paths do we have for x ?
    b. How many du-paths do we have to cover to get full **all-defs coverage** for x? When counting we just want to know how many of the du-paths should be covered; ignore the fact that some du-paths might include another. The same applies for question c below.
    c. How many du-paths do we have to cover to get full **all-uses coverage** for x? (tips: check first what "all-uses" coverage exactly means)

3. Consider now the following CFG, decorated with the def/use information with respect to three variables: x, y, and a.



a. How many du-paths do we have for each variable? Again, when counting we just want to know how many of the du-paths should be covered; ignore the fact that some du-paths might include another. The same applies for questions c and d below.
b. How many du-paths do we have to cover to get full **du-paths coverage** for y ?
c. How many du-paths do we have to cover to get full **all-uses coverage** for a?
d. How many du-paths do we have to cover to get full **all-defs coverage** for x?

4. Consider the program **COMMON** with the following header:

boolean **COMMON**(int[ ] a, int[ ] b)

The program takes two non-null arrays of integers. It returns true if the two arrays have an element in common, else false. Write a formal specification in terms of pre- and post-condition for this program. Use "retval" or "return" in your post-condition to refer to the program's return value.
We will need to use quantifiers to specify this program. There are two standard quantifiers you can use: **forall** (∀) and **exists** (∃). Few examples:

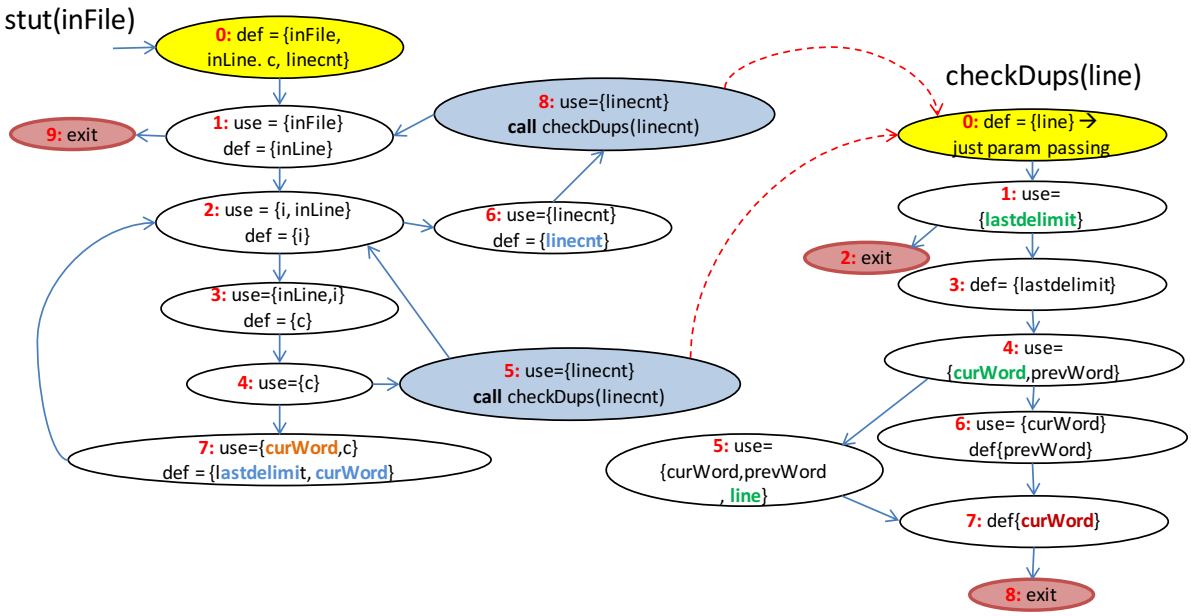
- To say that for all valid indices I of the array a, a[i] is 0, we write: (∀i: 0≤i<#a: a[i]=0). Or alternatively, (∀i:: 0≤i<#a ⇒ a[i]=0).
- (∃i: 0≤i<#a: a[i]=0) means that there exists one valid index i of the array a, such that a[i] is 0. We can also write it as (∃i:: 0≤i<#a /\ a[i]=0).

5. Consider the program SORT with the following header:

int[ ] **SORT**(int[ ] a)

The program takes a non-null integer array a containing **distinct** elements. The program returns a copy of a, but sorted in the increasing order. Write a formal specification in terms of pre- and post-condition for this program.

6. Consider the following CFGs of two methods. This is taken from the example class Stutter in O&A Section 2.5, Figure 2.34 and 2.35. The class is used to print repeated words in a file. The CFGs below belong to the methods **stut** and **checkDups**; the first contains calls the second. The CFGs have been decorated with the def/use information of various variables. Few notes first:
    a. The CFGs are as such that when a node contains both def and use of the same variable x, the use occurs before the def in the actual code that the node represents.
    b. The method checkDups returns void, but notice that it also affects instance variables of Stutter.



We will consider **three coupling variables** between **stut** and **checkDups**: **lastdelimit**, **curWord**, and **linecnt**. The last one requires some explanation. Notice that in both calls to **checkDups**, **stut** passes the value of **linecnt**. This parameter is passed by value to **checkDups**; so technically it is being copied to a local copy called **line** within **checkDups**. However, we will consider the first use of **line** to be the first use of **linecnt**.

a. How many coupling du-paths do we have for each coupling variables over the call site 8 ?
b. How many coupling du-paths do we have for each coupling variables over the call site 5 ?
c. How many coupling du-paths over call site 8 do we have to cover to get full "All Coupling Uses Coverage" (**ACUC**) over the variable **lastdelimit** ?
   How many (the same question as above) for call site 5?
d. How many coupling du-paths paths over call site 8 do we have to cover to get full ACUC over the variable **curWord**?
   How many (the same question as above) for call site 5?