Chris Cargile - March 7, 2014 CSCI658 - Testing and Maintenance

HOMEWORK ASSIGNMENT: CHAPTER 13

Question 13.1:

a) **Provide a precise definitions of the statements:**
- all p-use some c-use
- all c-use some p-use

Foremost the imprecise definitions and terminology required are restated as translated for my understanding, in the order in which the book gives them, below:
- For all definitions and uses => For all DU Pairs => DU_{pairs}
- Exercise all (def, c-use) pairs => exercise all DU_{pairs} inside or outside the predicate (ie: loop-control)
- Exercise all (def, p-use) pairs => exercise all DU_{pairs} inside the predicate statement

I defined the two criteria as:

- **(Formally)** All p-use some c-use:
A test suite *T* for a program *P* satisfies the 'all p-use, some c-use' criteria iff, for at least one DU use *use* in *P* and for each DU path *dp* of *P*, there exists at least one test case in *T* that exercises a DU pair that includes *use* and another that includes *dp.*
…AND…(In my own words) Within a section of evaluated code, when a definition-use pair is identified, for each definition, if the (corresponding) usage for the DU pair occurs within a predicate (loop control) statement, the usage will be executed at least once. In addition to exercising each predicate-located usage at least once, at least one exercising of the usage found not in the predicate statement will occur. Simply stated, this seems similar to achieving DU_{paths} except it expands on that by ensuring *at least* one additional non-predicate usage of each DU is executed one time at a minimum.

- **(Formally) All c-use some p-use:**
A test suite *T* for a program *P* satisfies the all p-use, some c-use criteria iff, for each use *use* of *P* and for **at least one** DU path *dp* of *P*, there exists at least one test case in *T* that exercises a DU pair that includes all *use*s and *dp.*
..and..
(In my own words) The meaning of the formal definition follows in line with the aforementioned concept's, except that for each DU pair, if there is a usage inside a predicate, it might be visited at least once. However, all usages not found within a predicate (ie: inside the body of each loop or sub-loop) inside the analyzed code segment will be executed at least once, will be visited for each DU pair.

b) **Describe the differences in the test suites derived applying the different criteria to function cgi_decode in Figure 13.1.**
Similarly to the all paths criterion, which assumes each predicate is evaluated at least once, we would need to achieve such a level of testing adequacy. Further, we would need to examine the level of coverage beyond statement or branch(/path) coverage to the level of definitions and usages, by examining DU pairs, to ensure each DU is assessed, appropriately.

- **All p-use some c-use:**
  Ensuring each predicate evaluates to true once and false once, at minimum, is not adequate for predicate criterion on the DU level. While a program being evaluated may have 0 LOC outside of predicate statements, the data use analysis should be the source to reveal such an occurrence. There will nearly always be at least one computational usage for a DU pair inside the predicate or its body, otherwise the predicate might evaluate to True forever! Thus, analyzing such a level of adequacy statically becomes a matter dealing in terms of determinability.
  Consider, for example, the below code segment:
  ```
  1.      p=-1;
  2.      while(p++){
  3.      if(p)
  4.          ..
  5.      }
  ```
  Sequentially executing this code would mean ensure the c-use part is ensured by line 1. The p-use section would begin to be ensured by line 2, but would fall short of being met due to the body of the while loop not being reached on the first pass. Thus the predicate at 2 would be reached but not that at 3. Therefore, to ensure the loop body of all nested loops is reachable, deterministically, there must be a definition-clear path inside the body of the outermost loop and within the body of any nested loops for which any additional definitions are made such that their definition is cleared within them somewhere.

  A test suite of this criterion would require the suite ensures:

  - *eptr==true; and *eptr==false
  - and while(*eptr==true), above, we would need to ensure each sub-predicate evaluates once to True once and to False once, recursively. This ensures the p-use part of the criterion is met. For each DU, we would need to ensure that it is exercised not in a predicate-usage to meet the c-use part of the criterion.

- **All c-use some p-use:**

Question: 13.2:
Demonstrate the subsumes relation between all p-use some c-use,all c-use some p-use, all DU pairs, all DU paths and all definitions.

Response:

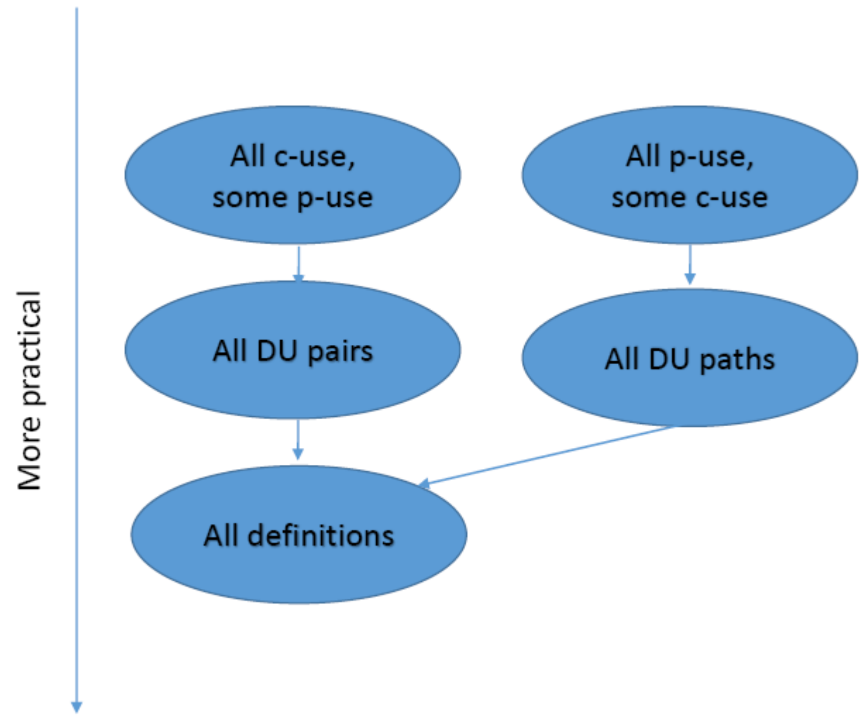


Figure: Subsumes relations for DU

Question 13.3:
How would you treat the buf array in figure 16.1's transduce() procedure?
Response:
It is a bit unclear to me what the answer to this question is, based on the book's vagueness or inadequacy in its general discussion of how to compute the number of DU paths, the definition of a definition-clear path of itself, and on how to 'treat' constructs. Specifically, the book on pg. 238, last full paragraph is vague and non-descriptive for these issues (it seems to me).

The intention regarding 'treatment' itself is unclear to me. From the text, it seems treating the buf array could be a matter of determining the possible DU paths/definition-clear paths OR determining the # possible paths for (ie: size of test suite needed to cover) a segment of a program. Admittedly, maybe I'm not understanding the material in this chapter/question, but it seems the appropriate 'treatment' is a matter of determining an adequately sized test suite, depending on the DU test adequacy criteria-level determined appropriate.

Optimally, I'd have provided a suitable test suite that guaranteed traversing the various DU paths. However, since I'm not sure how to do that, I provide, speculatively, what my process for doing so would have been, if I did understand how:

-not knowing the details of the emit() procedure's implementation, we would not know whether 'pos' is redefined outside the scope of the transduce() procedure, we would have to first limit the scope of our analysis to the procedure we can '*see'* to be practical. Then, although there could be many cases where the 'default' case is reached in the switch-control, there are only two cases for the non-default conditions to be true, all of which are exclusive to the others, giving us the available paths: a total of 3.

Still, understanding the ***DU** paths* remains another matter - still unclear to me.

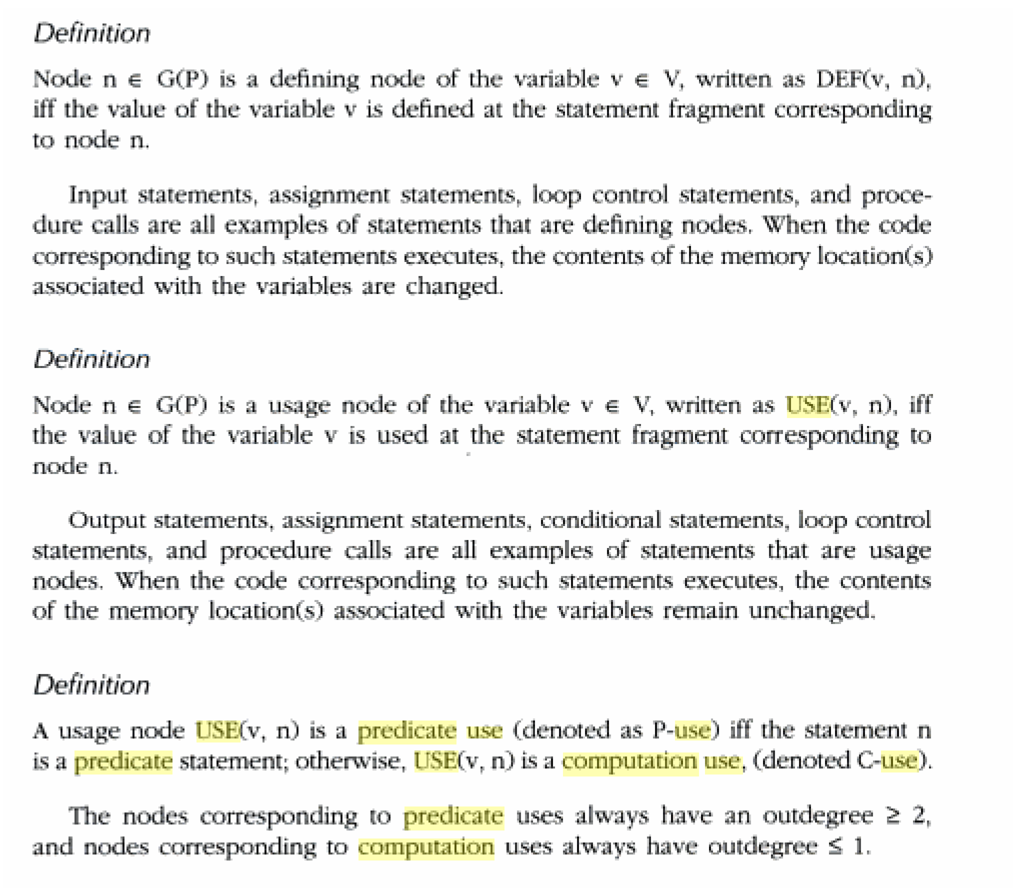
If 'treatment' concerns deriving a test suite, then a minimally -thorough/-sized test suite *T* might be comprised of at least 3 T_{Cases} (where the test suite represents standard input's contents) to reach all loop's cases according to the specified adequacy criterion, resembling:

- First understand the tests needed to enter all branches: {'\0', '#LF', '#CR'}
- Next, determine the means by which to execute each DU, i/a/w the adequacy criterion, such as by proposing a Test Suite for:
  $TC_{\text{DU pairs}}$;
  $TC_{\text{DU paths}}$; and
  $TC_{\text{DU def}}$

APPENDIX

(Question 1-A)'s use of the word 'predicate' was interpreted to mean the DU pair was located within an if() branch for control-flow and "computational" uses referred to DU pairs located inside the body or similar to aliases

Tthis interpretation is supported by information obtained by a google search, which afforded me a one-page synopsis of the terminology provided in {Software Testing: A Craftsman's Approach, Second Edition, by Jorgensen, P.C.}, captured below:



Definition

Node n ∈ G(P) is a defining node of the variable v ∈ V, written as DEF(v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node n ∈ G(P) is a usage node of the variable v ∈ V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.

Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node USE(v, n) is a predicate use (denoted as P-use) iff the statement n is a predicate statement; otherwise, USE(v, n) is a computation use, (denoted C-use).

The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have outdegree ≤ 1.

Figure. A captured summary of terms "p-use", "c-use" from Google query