Chapter 5, 8, 10, 12, & 13 Questions

Dr. Paul West

Department of Computer Science
College of Charleston

April 3, 2014

Problem 1

- We construct large. complex software systems by breaking them into manageable pieces. Likewise, models of software systems may be decomposed into more manageable pieces. Briefly describe how the requirements of model compactness, predictiveness, semantic meaningfulness, and sufficient generality apply to approaches for modularizing models of programs. Give examples.
-

Problem 1

- We construct large. complex software systems by breaking them into manageable pieces. Likewise, models of software systems may be decomposed into more manageable pieces. Briefly describe how the requirements of model compactness, predictiveness, semantic meaningfulness, and sufficient generality apply to approaches for modularizing models of programs. Give examples.
- Somewhat Subjective, but:
  - Compactness: Clarity of Model
  - Predictiveness: Accuracy to Software
  - Semantic Meaningfulness: Mapping to Tests/Software
  - Sufficient Generality: Not tying to any software technology

Problem 2

- Models are used in analysis, but construction of models from programs often requires some form of analysis. Why bother, then? If one is performing an initial analysis to construct a model to perform a subsequent analysis, why not just merge the initial and subsequent analysis and dispense with defining and constructing the model?
-

Problem 2

- Models are used in analysis, but construction of models from programs often requires some form of analysis. Why bother, then? If one is performing an initial analysis to construct a model to perform a subsequent analysis, why not just merge the initial and subsequent analysis and dispense with defining and constructing the model?
- A Model created from specifications is more closely related to what the user intended (requirements), then from software. Therefore, the Model will aid in testing the software.

Problem 5

- A directed graph is a set of nodes and directed edges. A Mathematical relation is a set of ordered pairs.
- 1) If we consider a directed graph as a representation of a relation, can we ever have two distinct edges from one node to another?
- 2) Each ordered pair in the relation corresponds to an edge in the graph. Is the set of nodes superfluous? In what case might the set of nodes of a directed graph be different from the set of nodes that appear in the ordered pairs?

Problem 5

- A directed graph is a set of nodes and directed edges. A Mathematical relation is a set of ordered pairs.
- 1) If we consider a directed graph as a representation of a relation, can we ever have two distinct edges from one node to another?
- 2) Each ordered pair in the relation corresponds to an edge in the graph. Is the set of nodes superfluous? In what case might the set of nodes of a directed graph be different from the set of nodes that appear in the ordered pairs?
- 1) Yes, A->B, B->A
- 2) Not necessarily–isolated nodes.

Problem 7

- Can the number of basic blocks in the control flow graph representation of a program ever be grater then the number of program statements? If so, how? If not, why not?
-

Problem 7

- Can the number of basic blocks in the control flow graph representation of a program ever be grater then the number of program statements? If so, how? If not, why not?
- Yes, when there are multiple control flows in one program statement (ternary operator). a < b ? c : d;

Problem 1

- We stated that finite state verification falls between basic flow analysis and formal verification in power and cost, but we also stated that finite state verification techniques are often designed to provide results that are tantamount to formal proofs of program properties. Are these two statements contradictory? If not, how can a technique that is less power than formal verification produce results that are tantamount to formal proofs?
-

Problem 1

- We stated that finite state verification falls between basic flow analysis and formal verification in power and cost, but we also stated that finite state verification techniques are often designed to provide results that are tantamount to formal proofs of program properties. Are these two statements contradictory? If not, how can a technique that is less power than formal verification produce results that are tantamount to formal proofs?
- No, as complete finite state verification can be equally as powerful as formal verification.

Problem 4

- A property like “if the button is pressed, then eventually the elevator willc ome” is classified as a liveness property. However, the stronger real-time version “if the button is pressed then the elevator will arrive within 30 seconds” is technically a safety property rather than a liveness property. Why?
-

Problem 4

- A property like “if the button is pressed, then eventually the elevator willc ome” is classified as a liveness property. However, the stronger real-time version “if the button is pressed then the elevator will arrive within 30 seconds” is technically a safety property rather than a liveness property. Why?
- Liveness is only violate given an “infinite length of execution”. Safety properties have a finite execution length.

Problem 1

- In the Extreme Programming (XP) methodology, a written description of a desired feature may be a single sentence, and the first step to designing the implementation of that feature is designing and implementing a set of test cases. Does this aspect of the XP methodology contradict our assertion that test cases are a formalization of specification?
-

Problem 1

- In the Extreme Programming (XP) methodology, a written description of a desired feature may be a single sentence, and the first step to designing the implementation of that feature is designing and implementing a set of test cases. Does this aspect of the XP methodology contradict our assertion that test cases are a formalization of specification?
- No, since the desired feature is now our requirement/formal specification.

Problem 3

- Identify independently testable units in the following specification.
- There are many acceptable answers...
  3+5
  12.0%5 1/0
  etc...

Problem 1

- Let us consider the following loop, which appears in C lexical analyzers generated by the tool flex.

```
for(n = 0; n < max\_size && (c = getc(yyin)) != EOF && c != '\n', ++n){
    buf[n] = (char)c;
}
```

  Device a set of test cases that satisfy the compound condition adequacy criterion and a set of test cases that satisfy the modified condition adequacy criterion with respect to this loop.

-

Problem 1

- Let us consider the following loop, which appears in C lexical analyzers generated by the tool flex.

```
for(n = 0; n < max\_size && (c = getc(yyin)) != EOF && c != '\n', ++n){
    buf[n] = (char)c;
}
```

  Device a set of test cases that satisfy the compound condition adequacy criterion and a set of test cases that satisfy the modified condition adequacy criterion with respect to this loop.
- T1=(n = max_size, c = 'a')
- T2=(n = max_size, c = EOF)
- T3=(n = max_size, c = '\n')
- T4=(n = 0, c = 'a')
- T5=(n = 0, c = '\n')
- T6=(n = 0, c = EOF) $a =$ "n <max_size", $b =$"c!=EOF", $d =$"c!=\n".

Problem 1

- Let us consider the following loop, which appears in C lexical analyzers generated by the tool flex.

```
for(n = 0; n < max\_size && (c = getc(yyin)) != EOF && c != '\n', ++n){
    buf[n] = (char)c;
}
```

  Device a set of test cases that satisfy the compound condition adequacy criterion and a set of test cases that satisfy the modified condition adequacy criterion with respect to this loop.

- T1=(n = max_size, c = 'a')
- T4=(n = 0, c = 'a')
- T5=(n = 0, c = '\n')
- T6=(n = 0, c = EOF)

Problem 2

- Consider:

```
if (pos < parseArray.length && (parseArray[pos] == '{' || parseArray[pos] == '}' || pars
    continue;
}
```

- a) Derive a set of test case specifications and show that is satisfies the MC/DC criterion for this statement. Let Room = pos < parseArray.length, Open = $parseArray[pos] == "\{"$, Close = $parseArray[pos] == "\}"$, Bar = $parseArray[pos] == "\{"$
- b) Do the requirements for compound condition coverage and modification condition/decision coverage differ in this case? Aside from increasing the number of test cases, what difference would it make if we attempted to exhaustively cover all combinations of truth values for the basic conditions?
-

Problem 2

- Consider:

```
if (pos < parseArray.length && (parseArray[pos] == '{' || parseArray[pos] == '}' || pars
    continue;
}
```

- a)

	Room	Open	Close	Bar
1	false	-	-	-
2	true	false	false	false
3	true	true	-	-
4	true	-	true	-
5	true	-	-	true

Problem 2

- Consider:

```
if (pos < parseArray.length && (parseArray[pos] == '{' || parseArray[pos] == '}' || pars
    continue;
}
```

- Some difference, our instance of MC does not test Room == true, Open = true, Close = false, Bar = true. For testing, it doesn't make a difference due to short-circuiting.

Problem 4

- The number of basis paths (cyclomatic complexity) does not depend on whether nodes of the control flow graph are individual statements or basic blocks that may contain several statements. Why?
-

Problem 4

- The number of basis paths (cyclomatic complexity) does not depend on whether nodes of the control flow graph are individual statements or basic blocks that may contain several statements. Why?
- The number of paths is independent of what the nodes of a graph contain. The basis paths only rely on paths through the nodes not it's contents.

Problem 6

- If the modified condition/decision adequacy criterion requires a test case that is not feasible because of interdependent basic conditions, should this always be taken as an indication of a defect in design or coding? Why/Why not?
-

Problem 6

- If the modified condition/decision adequacy criterion requires a test case that is not feasible because of interdependent basic conditions, should this always be taken as an indication of a defect in design or coding? Why/Why not?
- Total Opinion:
  Neither: defense in depth
  Both: Dead code, more design

Problem 1

- Sometimes a distinction is made between uses of values in predicates (p-uses) and computation uses in statement (c-uses).
  a) What is all c-use and what does all p-use mean?
  b) Describe the different in test suites derived from applying c-uses/p-uses to cgi-decode.
-

Problem 1

- Sometimes a distinction is made between uses of values in predicates (p-uses) and computation uses in statement (c-uses).
  a) What is all c-use and what does all p-use mean?
  b) Describe the different in test suites derived from applying c-uses/p-uses to cgi-decode.
- All c-use requires that for each definition of a variable x, and each c-use of x is reachable from the definition. IE: execute all computations that are affected by the definition of a variable, contain a def-clear path.
- All p-use requires that for each definition of a variable x, and each p-use of x reachable from the definition, contain a def-clear path. IE: test all branches affected by the definition of a variable.

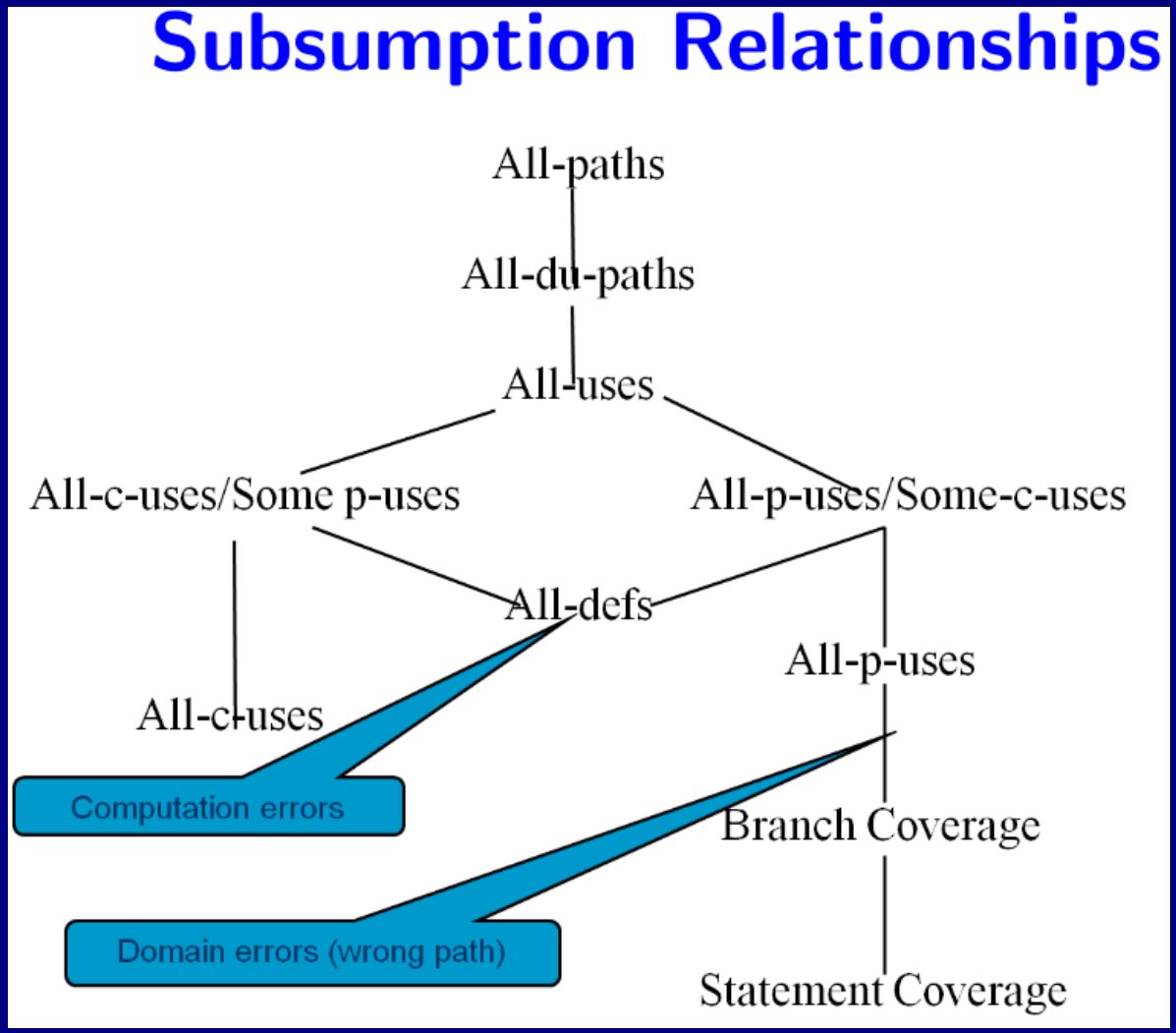
Problem 1

- Sometimes a distinction is made between uses of values in predicates (p-uses) and computation uses in statement (c-uses).
  a) What is all c-use and what does all p-use mean?
  b) Describe the different in test suites derived from applying c-uses/p-uses to cgi-decode.
- b) c-use would test the computations like line 31: $*dptr = 16 * digit\_hight + digit\_low$, while p-use would test the branches (if/while statementes).

Problem 2

- Demonstrate the subsume relation between all p-use, all c-use, all DU pairs, all DU paths and all definitions.

Problem 2


Subsumption Relationships
All-paths
All-du-paths
All-uses
All-c-uses/Some p-uses
All-p-uses/Some-c-uses
All-defs
All-p-uses
All-c-uses
Computation errors
Branch Coverage
Domain errors (wrong path)
Statement Coverage

Problem 3

- How would you treat the buf array in the transduce procedure shown in Figure 16.1?
-

Problem 3

- How would you treat the buf array in the transduce procedure shown in Figure 16.1?
- Like the C function it is... Okay, DU-pairs: Test strings with a mixture of any number of LF, CR, with other “normal” characters.