CS 130 SOFTWARE ENGINEERING

MODERN SYSTEMATIC TESTING

Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim

SYSTEMATIC AND RANDOM TESTING

SYSTEMATIC VS. RANDOM TESTING

- Random sampling means we can automate and apply a very large number of tests but even then the coverage will remain very small (particularly for complex problems).
- For example, in the case of buffer overrun failure, the likelihood of adding a very long sequence of elements is very small.
- So faults with small profiles and the size of input spaces force a hybrid where we must consider some systematic testing – possibly reinforced with randomised testing.

SYSTEMATIC VS. RANDOM TESTING

- Key take away:
  - Be systematic in exploring search space but randomize to explore variation.
  - Prioritize boundary conditions first
  - Array=nulll
  - Each array is inititalized
  - Accessing a range out of bound

WHITE BOX VS. BLACK BOX

- When we consider details of the implementation, it’s known as “white box testing”
- When we work from external descriptions, treating the implementation as an opaque artifact with inputs and outputs: it’s called “black box testing.”

WHITE BOX STRUCTURAL TESTING IS BETTER

- Testing that is based on the structure of the program.
- Usually better for finding defects than for exploring the behavior of the system as a black-box.
- Fundamental idea is that of "basic block" and flow graph – most work is defined in those terms.

TWO KINDS OF STRUCTURAL TESTING

- **Control oriented:** how much of the control aspect of the code has been explored?
  - This is what we studied: Statement coverage, Branch coverage, Path Coverage
- **Data oriented:** how much of the definition/use relationship between data elements has been explored.

RECAP: STRUCTURAL COVERAGE

- **Statement adequacy**
  - All statements have been executed by at least one test.
- **Branch Condition adequacy**
  - Let T be a test suite for program P. T covers all the basic conditions of P iff each basic condition of P evaluates to true under some test in T and evaluates to false under some test in T.
- **Path adequacy**
  - Let T be a test suite for program P. T satisfies the path adequacy criterion for P iff for each path p of P there exists at least one testcase in T that causes the execution of p.

MODEL BASED TESTING

THINK-PAIR-SHARE

- We learned white-box structural testing criteria (e.g., statement, branch, path coverage). How can you test programs as a black box?

- Enumerate test search space based on **specifications** and **abstract models.**

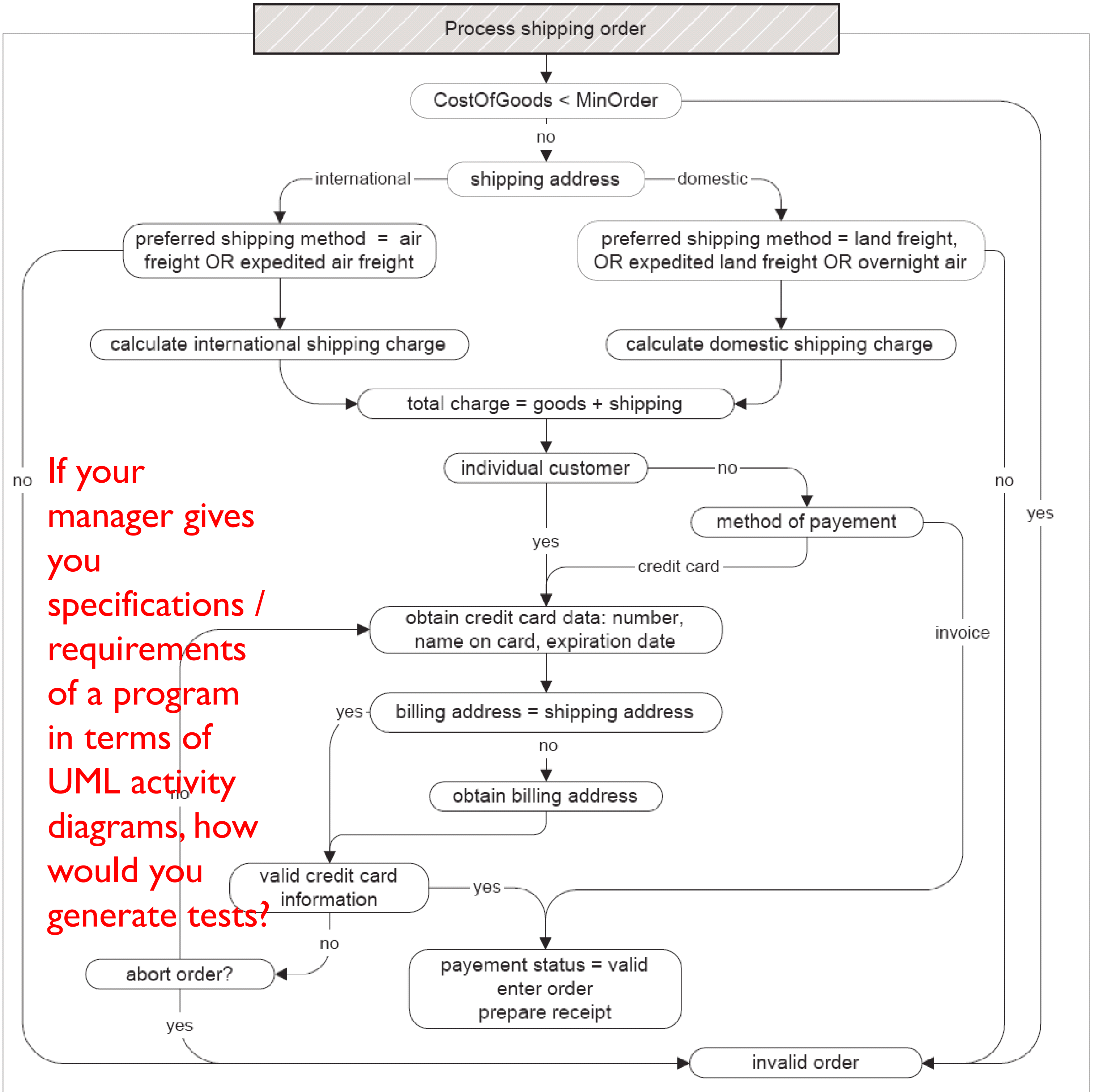
MODEL BASED TESTING

- We have some model of the system and use that to decide how to exercise the system. Typical examples of models include:
  - Decision trees/graphs
  - Workflows
  - Finite State Machines
  - Grammars
- All of these models provide **some kind of abstraction** of the system's behavior.

TYPE 1. UML ACTIVITY DIAGRAMS

- Often specify the human process the system is intended to support.
- Can be used to represent both “normal” and “erroneous” behaviors (and recovery behavior).
- Abstract away from internal representations.
- Focus on interactions with the system
- Similar to Control Flow Graph

If your manager gives you specifications / requirements of a program in terms of UML activity diagrams, how would you generate tests?

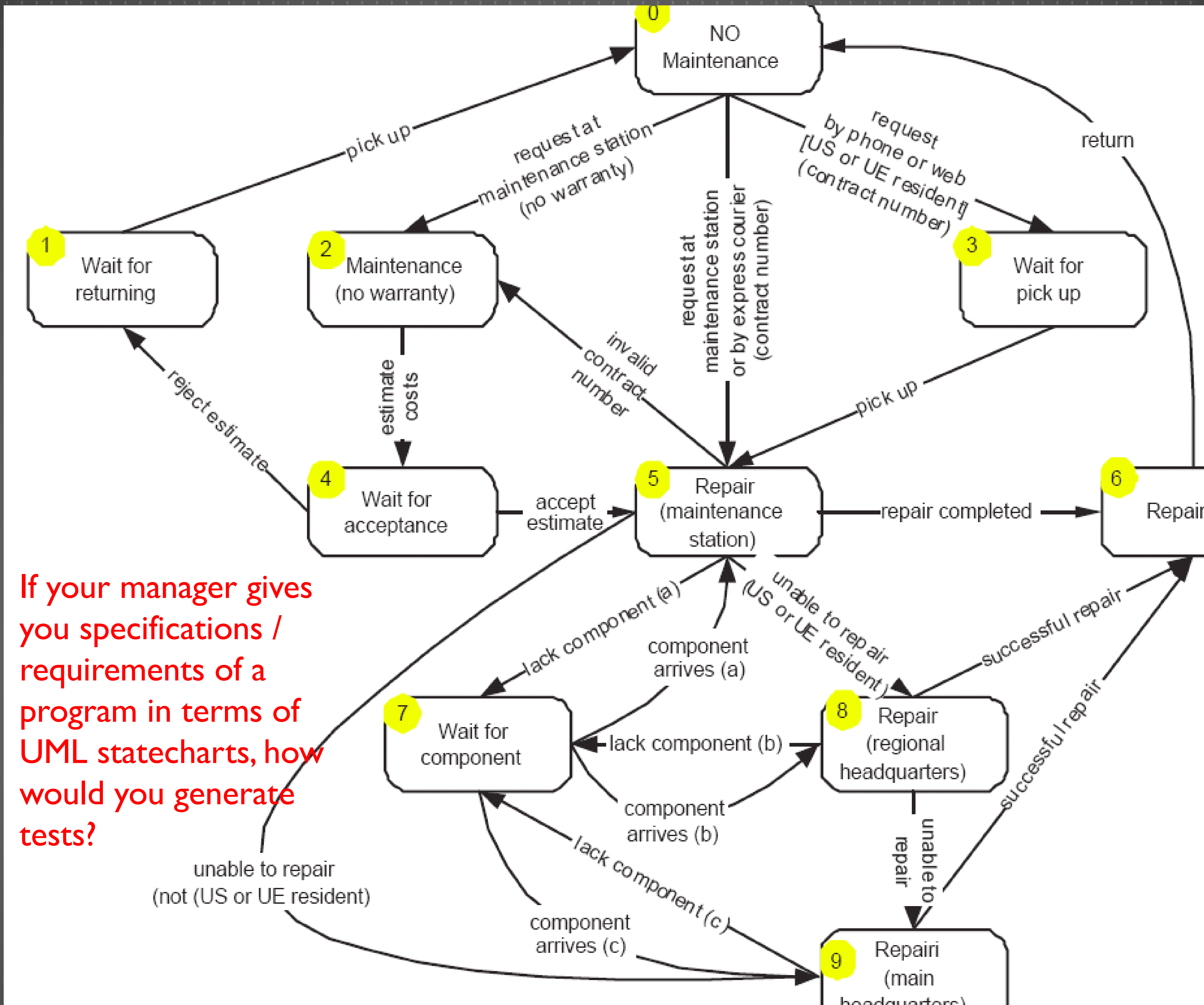


DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- **Node coverage** – ensure that test cases cover all the nodes in the activity diagram.
- **Branch coverage** – ensure we branch in both directions at each decision node.
- **Mutations** – we might also consider introducing mutations where the user does not follow the activity diagram

TYPE 2: FINITE STATE MACHINE

- Good at describing interactions in systems with a small number of modes.
- Good examples are systems like communication protocols or many classes of control systems (e.g. automated braking, flight control systems).



If your manager gives you specifications / requirements of a program in terms of UML statecharts, how would you generate tests?

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- Single state path coverage: collection of paths that cover the states:
- Single transition path coverage: collection of paths that cover all transitions.
- Boundary interior loop coverage: criterion on number of times loops are exercised.

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- We can consider **mutation** to discover how the system responds to unexpected inputs.
- We can use **probabilistic automata** to represent distributions of inputs if we want to do randomized testing.

TYPE 3: GRAMMAR BASED TESTING

- Grammars are used to describe well-formed inputs to systems.
- We can use grammars to generate sample inputs.
- We can use coverage criteria on a test set to see that all constructs are covered.

GRAMMAR BASED TESTING

$$
\begin{aligned}
\langle search \rangle &::= \langle search \rangle\ \langle binop \rangle\ \langle term \rangle \mid \boxed{\texttt{not}}\ \langle search \rangle \mid \langle term \rangle \\
\langle binop \rangle &::= \boxed{\texttt{and}} \mid \boxed{\texttt{or}} \\
\langle term \rangle &::= \langle regexp \rangle \mid \boxed{\texttt{(}}\ \langle search \rangle\ \boxed{\texttt{)}} \\
\langle regexp \rangle &::= Char\ \langle regexp \rangle \mid Char \mid \boxed{\texttt{\{}}\ \langle choices \rangle\ \boxed{\texttt{\}}} \mid \boxed{\texttt{*}} \\
\langle choices \rangle &::= \langle regexp \rangle \mid \langle regexp \rangle\ \boxed{\texttt{,}}\ \langle choices \rangle
\end{aligned}
$$

If your manager gives you specifications of search terms in terms of context free grammar, how would you generate tests?

DIFFERENT ADEQUACY CRITERIA ARE APPLICABLE

- Every production at least once
- Boundary conditions on recursive productions – 0, 1, many
- Probabilistic CFGs allow us to prioritize heavily used constructs.
- Probabilistic CFGs can be used to capture and abstract real world data.

MUTATION TESTING

THINK-PAIR-SHARE

- How would you test your own confidence about the adequacy of tests you have written?

WHAT IS MUTATION TESTING?

- **Mutation testing** is a structural testing method aimed at assessing/improving the **adequacy** of test suites, and estimating the number of faults present in systems under test.

WHAT IS MUTATION TESTING?

- The process, given program P and test suite T, is as follows:
  - We systematically apply mutations to the program P to obtain a sequence P1, P2,... Pn of mutants of P. Each mutant is derived by applying a single mutation operation to P.
  - We run the test suite T on each of the mutants, T is said to kill mutant Pj if it detects an error.
  - If we kill k out of n mutants the adequacy of T is measured by the quotient k/n, which is called a mutant killing ratio. T is mutation adequate if k=n.
- One goal of mutation testing is to assess or improve the **efficacy** of test suites in discovering defects.

KINDS OF MUTATIONS

- **Value Mutations:** these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.
- **Decision Mutations**: this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs. E.g. a typical mutation might be replacing a > by a < in a comparison.
- **Statement Mutations:** these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

VALUE MUTATION

- Here we attempt to change values to reflect errors in reasoning about programs.
- Typical examples are:
  - Changing values to one larger or smaller (or similar for real numbers).
  - Swapping values in initializations.
- The commonest approach is to change constants by one in an attempt to generate a one-off error (particularly common in accessing arrays).

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have t length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

DECISION MUTATION

- Modeling "one-off" errors by changing < to <= or vice versa (this is common in checking loop bounds).
- Modeling confusion about larger and smaller, so changing > to < or vice versa.
- Getting parenthesis wrong in logical expressions e.g. mistaking precedence between && and ||

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to t[i]>u will cause miscounting

We can model “one-off” errors in the loop bound by changing this condition to i<=t.length – provided array bounds are checked exactly this will provoke an error on every execution.

STATEMENT MUTATION

- Typical examples include:
  - Deleting a line of code
  - Duplicating a line of code
  - Permuting the order of statements.
- Coverage Criterion: We might consider applying this procedure to each statement in the program (or all blocks of code up to and including a given small number of lines).

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

THINK-PAIR-SHARE

- *Is mutation an appropriate tool for testing experiments? ICSE 2005, Andrews J.H et al.*
  - The most influential paper award at ICSE 2015
- They did a systematic study and found that mutation faults can effectively emulate real world faults.
- *Are Mutants a Valid Substitute for Real Faults in Software Testing? FSE 2014, Just et al.*

RECAP (1)

- Be systematic in exploring search space but randomize to explore variations.
- Generally enumerating all possible combinations is exhaustive but probably infeasible given cost constraints.
- Model based testing uses abstract models to effectively explore test search space.

RECAP (2)

- Mutations model low level errors in the mechanical production process.
- Mutation testing is a structural testing method aimed at assessing/improving the adequacy of test suites, and estimating the number of faults present in systems under test

QUESTIONS?