Software Testing

Part 4 of 4

Path Testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements with conditions are therefore nodes in the flow graph

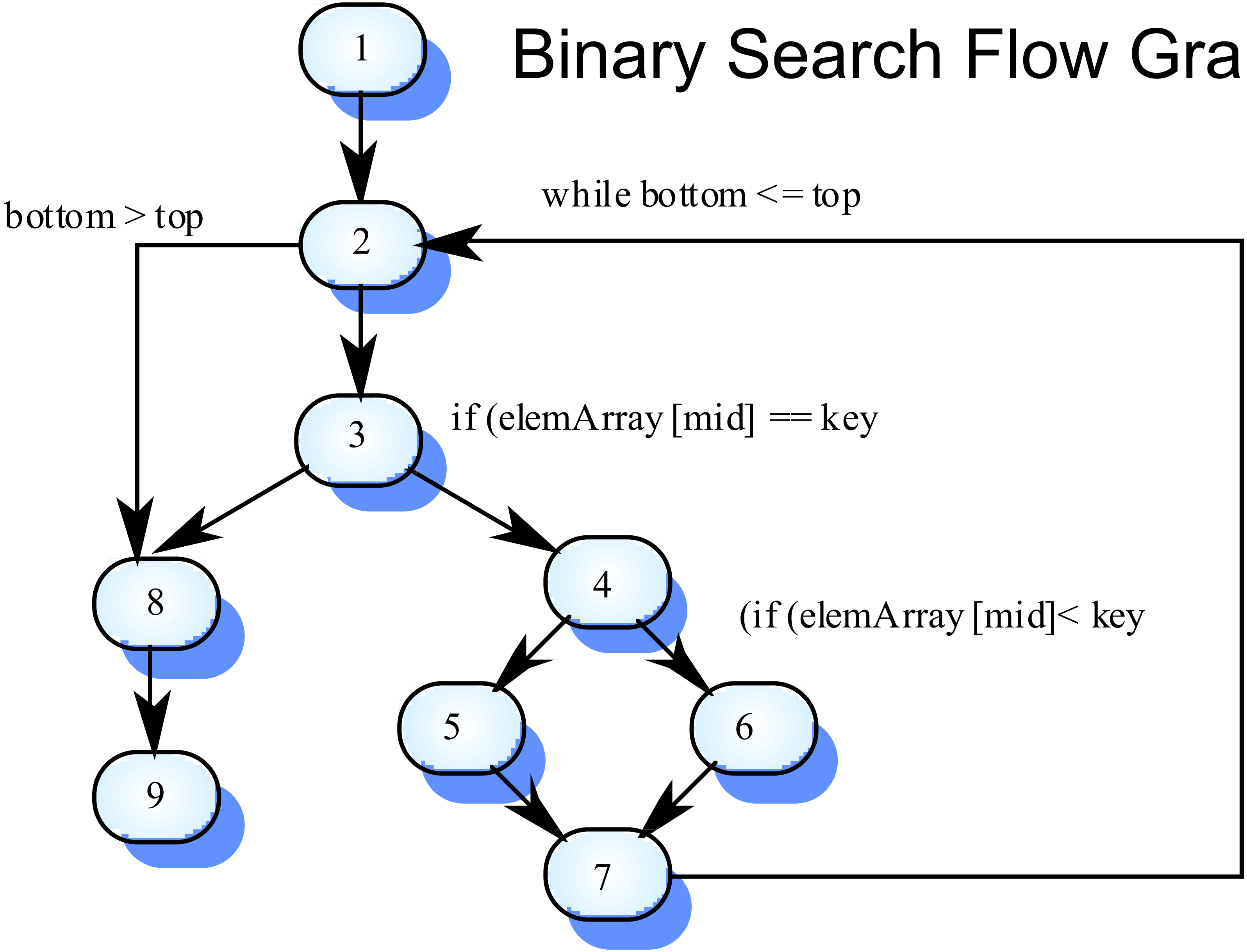
Program Flow Graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- Used as a basis for computing the cyclomatic complexity
- Cyclomatic complexity = Number of edges - Number of nodes +2

Cyclomatic Complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing
- Although all paths are executed, all combinations of paths are not executed

Binary Search Flow Graph



Independent Paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyzer may be used to check that paths have been executed

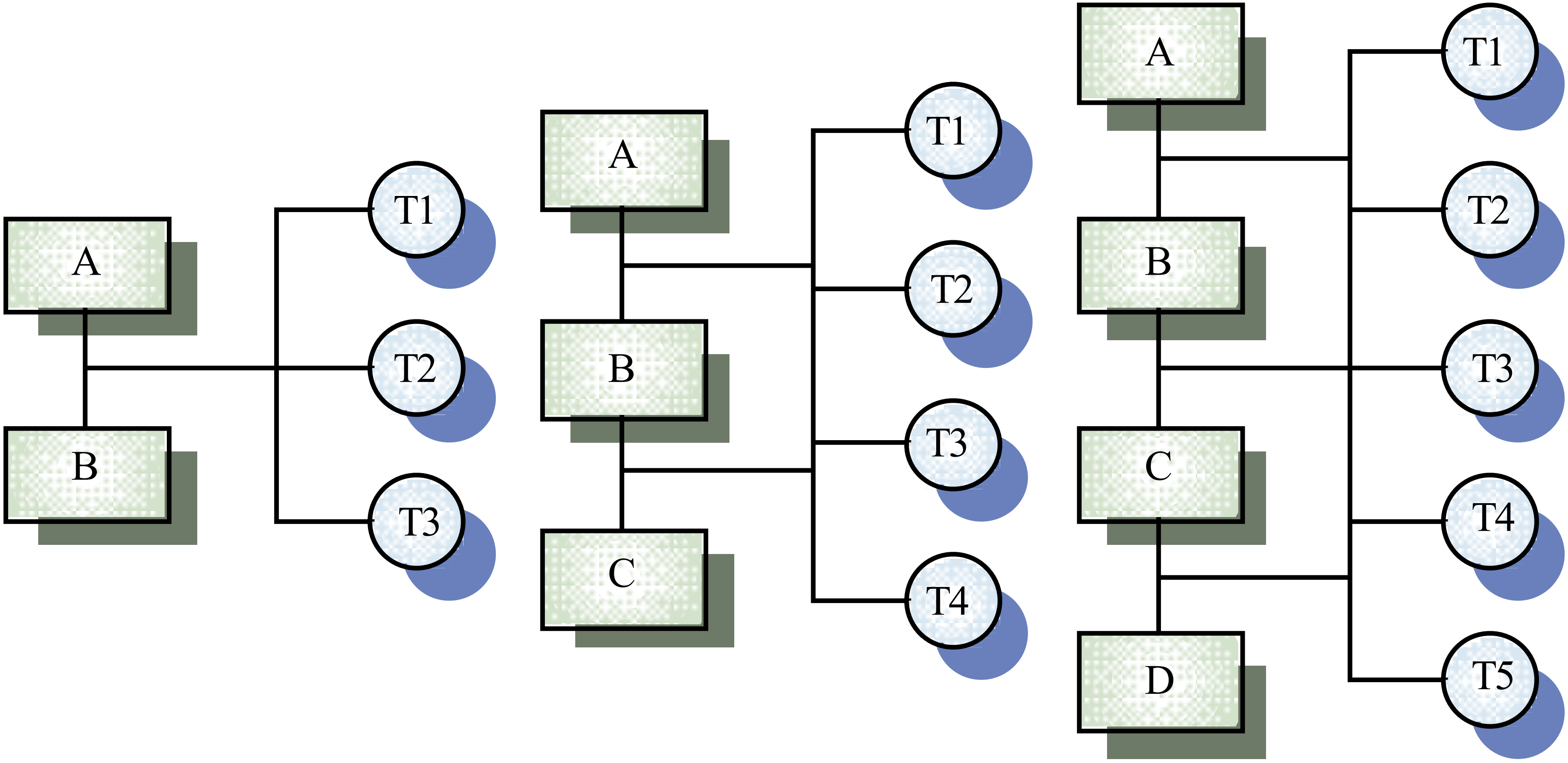
Feasibility

- Pure black box testing (specification) is realistically impossible because there are (in general) too many test cases to consider.
- Pure testing to code requires a test of every possible path in a flow chart. This is also (in general) infeasible. Also every path does not guarantee correctness.
- Normally, a combination of Black box and Glass box testing is done.

Integration Testing

- Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem

Incremental integration testing



Test sequence 1

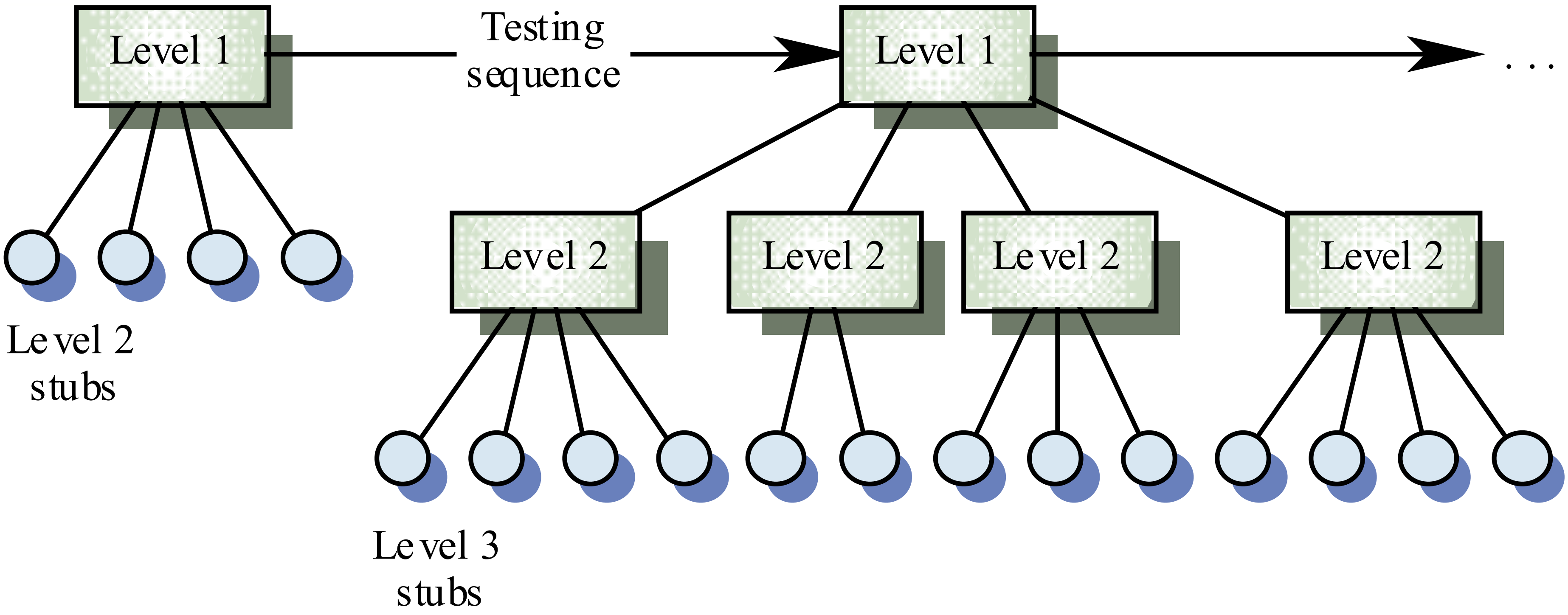
Test sequence 2

Test sequence 3

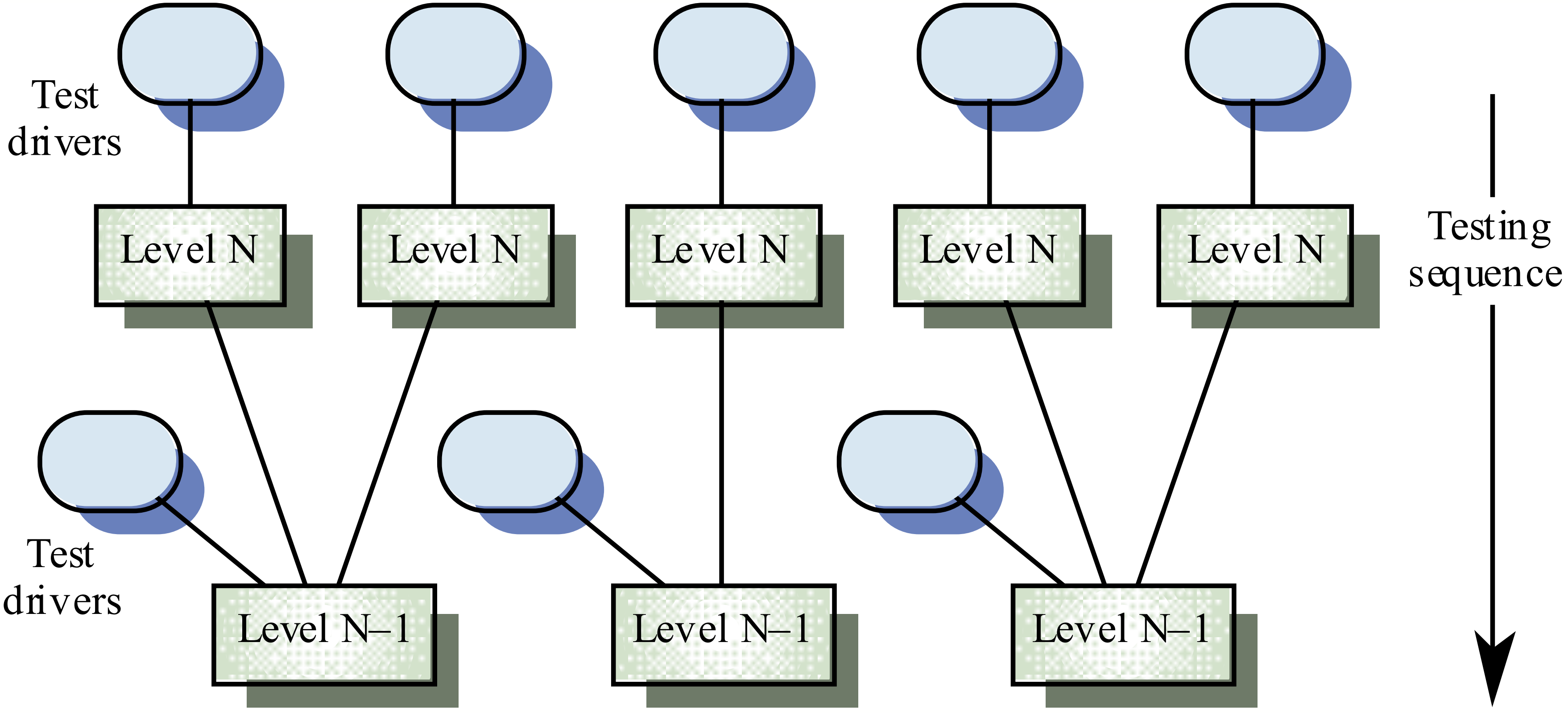
Approaches to Integration Testing

- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down Testing



Bottom-up Testing



Software Testing Metrics

- Defects rates
- Errors rates
- Number of errors
- Number of errors found per person hours expended
- Measured by:
  - Individual, module, during development
- Errors should be categorized by origin, type, cost

More Metrics

- Direct measures - cost, effort, LOC, etc.
- Indirect Measures - functionality, quality, complexity, reliability, maintainability

- Size Oriented:
  - Lines of code - LOC
  - Effort - person months
  - errors/KLOC
  - defects/KLOC
  - cost/KLOC

Proofs of Correctness

- Assertions, preconditions, post conditions, and invariants are used
- **Assertion** – something that is true at a particular point in the program
- **Pre conditions** must be true before something is executed
- **Post conditions** are true after something has executed
- **Invariants** are always true with a give scope (e.g., construct, loop, ADT)

Logical Properties

- Assertions describe the logical properties which hold at each statement in a program

- Assertions can be added to each line to describe the program

- Utilize a formal approach (e.g., first order predicate calculus, Z, spec#, etc.)

Example

```
//PRE: n in {1,2,3…}
int k, s;
int y[n];
k=0;
//ASSERT: k==0
s=0;
//ASSERT: s==0 && k==0
//LOOP INV: (k<=n) && (s==y[0]+y[1]+…+y[k-1])
While (k<n)
{
  //ASSERT: (k<n) && (s==y[0]+y[1]+…+y[k-1])
  s=s+y[k];
  //ASSERT: (k<n) && (s==y[0]+y[1]+…+y[k])
  k=k+1;
  //ASSERT: (k<=n) && (s==y[0]+y[1]+…+y[k-1])
}
//POST: (k==n) && (s==y[0]+y[1]+…+y[n-1])
```

Proving the Program

- Prove correct based on the loop invariant
- Use induction

- Basis:
  - Before loop is entered
  - k=0 and s=0 therefore
  - s=y[0-1]=y[-1]=0
  - Also k<=n since n in {1,2,3,…}

Using Induction

- Inductive Hypothesis
  - Assume for some k>=0,
  - s = y[0]+y[1]+…y[n-2]+y[n-1]
  - when ever n<=k
- Inductive step show s = y[0]+y[1]+…y[n-2]+y[n-1] is true for k+1
  - s = y[0]+y[1]+…+y[k+1-2]+y[k+1-1]
  - s = y[0]+y[1]+…+y[k-1]+y[k]
  - s = (y[0]+y[1]+…+y[k-1]) + y[k] Q.E.D

Proving can be Problematic

- Mathematical proofs (as complex and error prone as coding)
- Need tool support for theorem proving

- Leavenworth ‘70 did an informal proof of correctness of a simple text justification program. (Claims it’s correct!)
- London ‘71 found four faults, then did a formal proof. (Claims it’s now correct!)
- Goodenough and Gerhar ‘75 found three more faults.

- Testing would have found these errors without much difficulty

Automated Testing Tools

- Code analysis tools

- Static analysis
  - No execution
- Dynamic analysis
  - Execution based

Static Analysis

- Code analyzers: syntax, fault prone
- Structure checker
  - Generates structure graph from the components with logical flow checked for structural flaws (dead code)
- Data analyzer – data structure review. Conflicts in data definitions and usages
- Sequence checker – checks for proper sequences of events (open file before modify)

Dynamic Analysis

- Program monitors record snapshot of the state of the system and watch program behaviors
- List number of times a component is called (profiler)
- Path, statement, branch coverage
- Examine memory and variable information

Test Execution Tools

- Capture and replay
  - Tools capture keystrokes, input and responses while tests are run
  - Verify fault is fixed by running same test cases
- Subs and drivers
- Generate stubs and drivers for integration testing
  - Set appropriate state variables, simulate key board input, compare actual to expected
  - Track paths of execution, reset variables to prepare for next test, interact with other tools

Test Execution Tools

- Automated testing environments
- Test case generators
  - Structural test case generators based on source code – path or branch coverage
  - Data flow