Testing

Role in Unified Approach

Coverage:

- Structural/Coverage
- Model Based
- Test Generation from Model Checking (project)

Interaction of Coverage/Model Based Testing

Will Not Cover

- Statistical Methods
- Partition Methods
- Functional Testing

Role of Testing

Most Accessible and Common Method of V&V

Thorough testing should precede application of formal methods.

Some properties may be rigorously verified by testing.

(particularly at the component level)

Interaction and Relationships with Other V&V Methods

- Functional testing may (should) based on property specifications
- Structural/Coverage testing based on static analysis
- Model checking can be used for test generation
- Model checking and testing are a continuum
- Runtime monitoring is continuous testing
- Open Issues:

  Derivation of structural/coverage tests from property specifications.

  Unification of model-based and coverage testing

Component/Unit Test

Requires precise specification at component level.

Functionality defined as properties or pre-conditions/post-conditions.

Pre-conditions (test cases) must be defined

Exceptions to preconditions must be defined

Coverage tests may be readily derivable.

Oracle Problems

Post-Condition verifiers (Oracles) must be constructed

Complete oracle is correct implementation!

Common oracles are not complete.

Most oracles are human inspectors

Oracles for specific properties??

Coverage Analyses

Control Flow

- Statement coverage
- Decision coverage
- Condition coverage

  single/multiple
- Condition/Decision coverage

  variants of C/D coverage
- Path coverage

Data Flow

Use/Def relations

coverage (other)

- Function coverage
- Call coverage
- Loop
- Race
- Mutation coverage
- Table coverage
- Relational operator coverage

Structural/Coverage Testing

Establishes that a given execution "covers" some set of program structures or functions.

Why useful?

- Errors are likely to arise from control flow.
- Errors are likely to arise from widely separated definition and use of variables

Challenges

- Generating test cases conforming to coverage cases
- Cost of creating test cases

Issues:

Integration of property specification and coverage specification.

Construction of property specific coverage, abstraction and state space specification.

Combining abstraction with coverage testing

Role of Design in Testing

Formal model for component

- Components with precise definitions
- Implementation should follow model

Simple control structures

- State machine structure

Prescribed ranges for variables

Web Resources

http://www.testing.com/

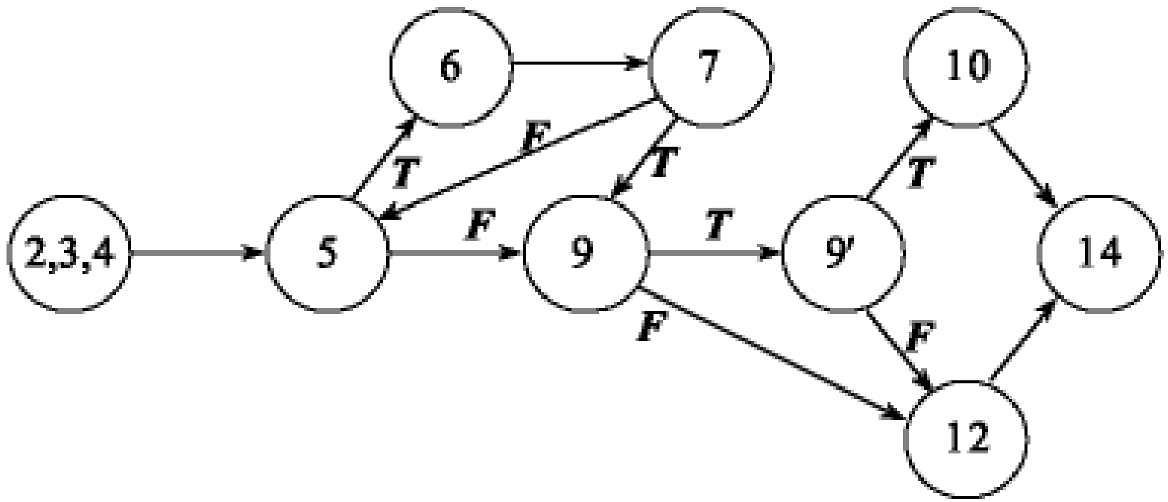
http://www.bullseye.com/

http://www.codecoveragetools.com/

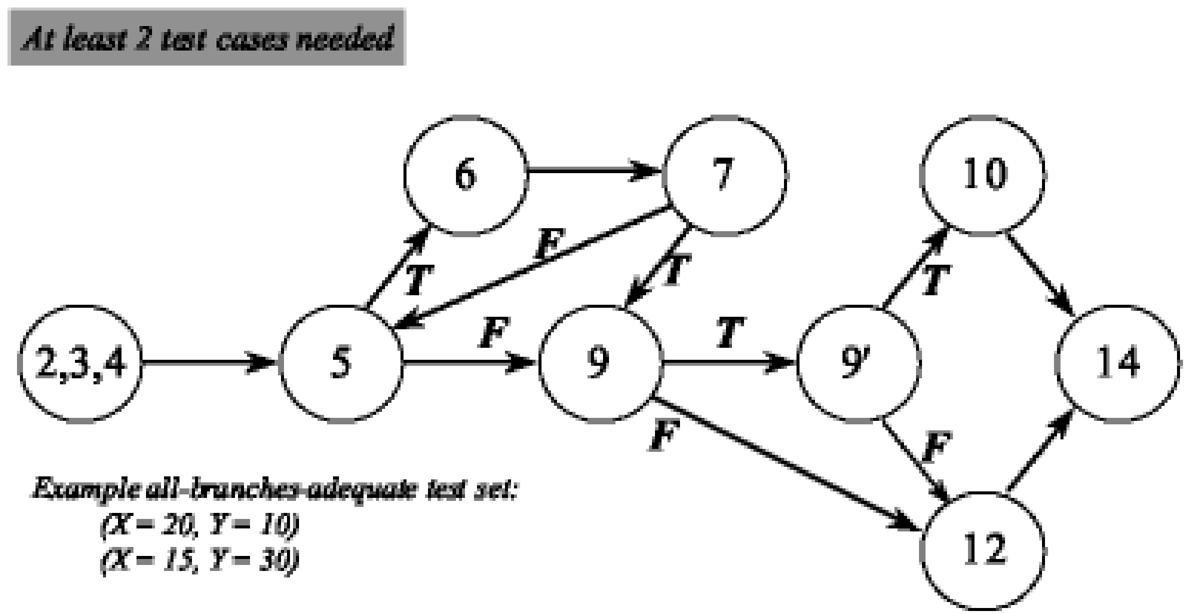
http://www.semdesigns.com/Products/TestCoverage/CTestCoverage.html

```
function P return INTEGER is
begin
        X, Y: INTEGER;
        READ(X); READ(Y); -- definition of X and Y
        while (X > 10) loop
                X := X – 10;
        exit when X = 10;
        end loop;
        if (Y < 20 and then X mod 2 = 0) then-- “short circuit” and operator
                Y := Y + 20;
        else
                Y := Y – 20;
        end if;
        return 2 * X + Y;
end P;
```

P's Control Flow Graph (CFG)

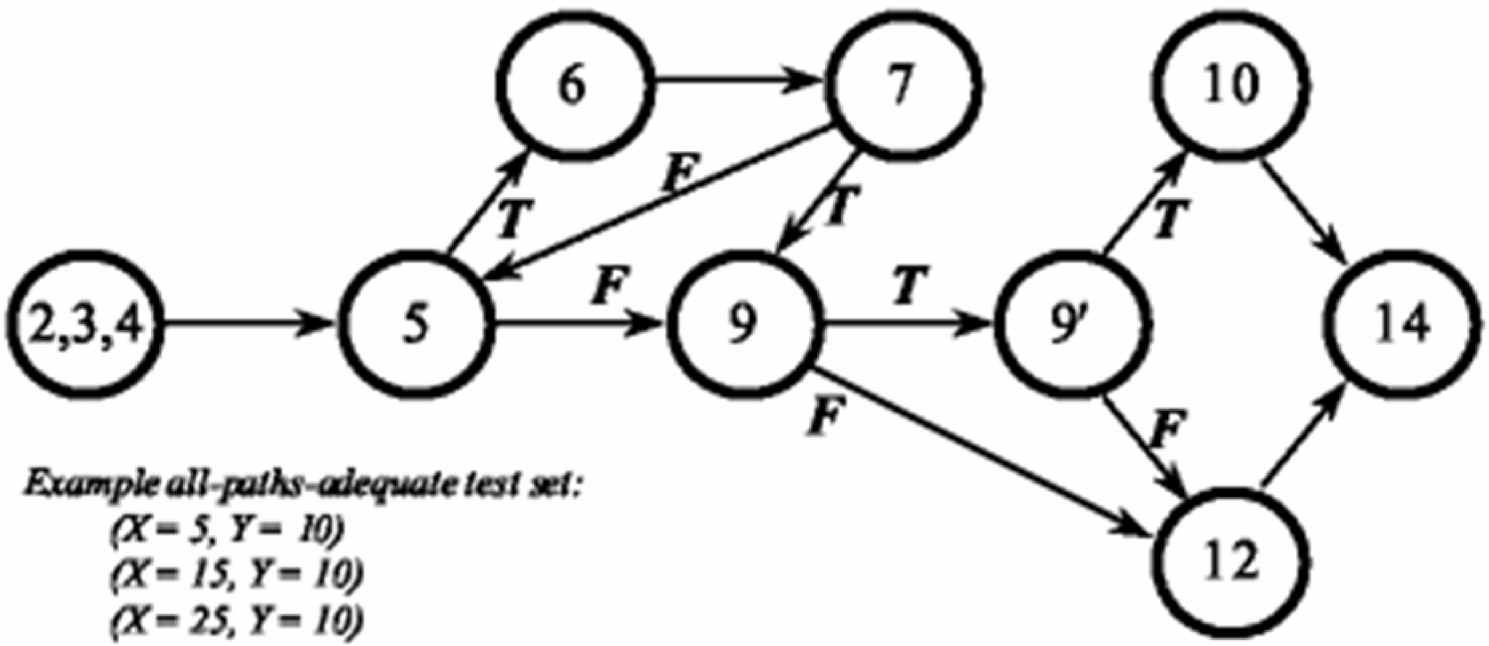


Branch Coverage of P



Path Coverage of P

Infinitely many test cases needed

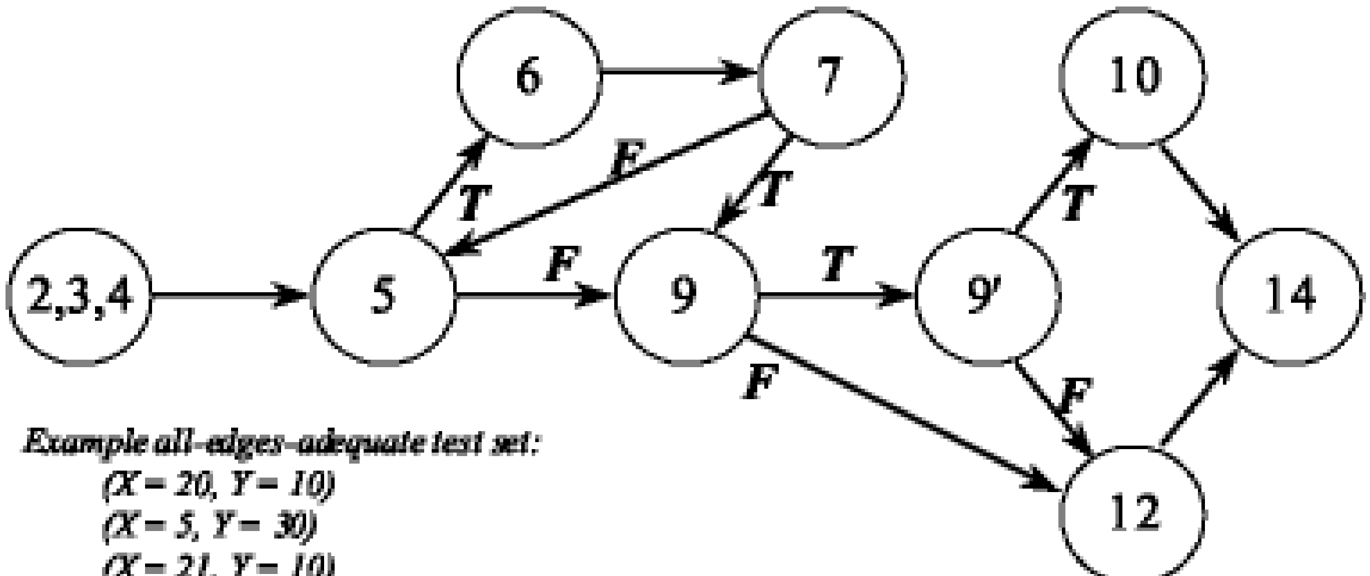


Example all-paths-adequate test set:

(X = 5, Y = 10)
(X = 15, Y = 10)
(X = 25, Y = 10)
(X = 35, Y = 10)
...

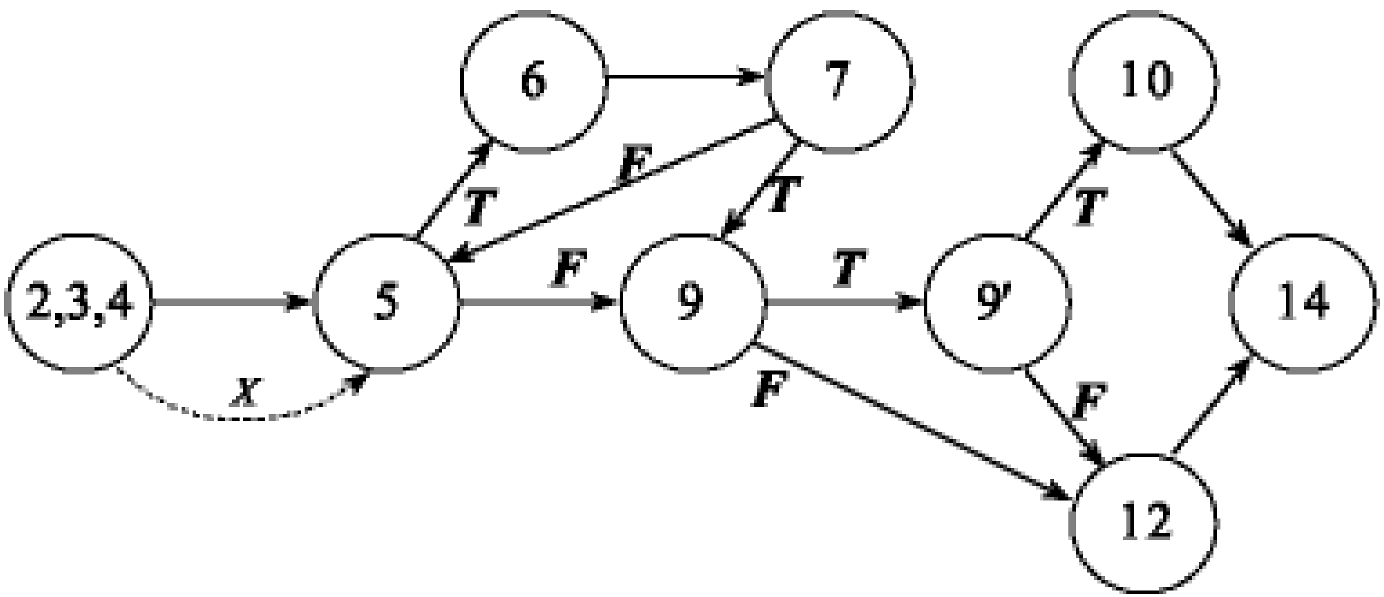
Condition Coverage of P

At least 3 test cases needed

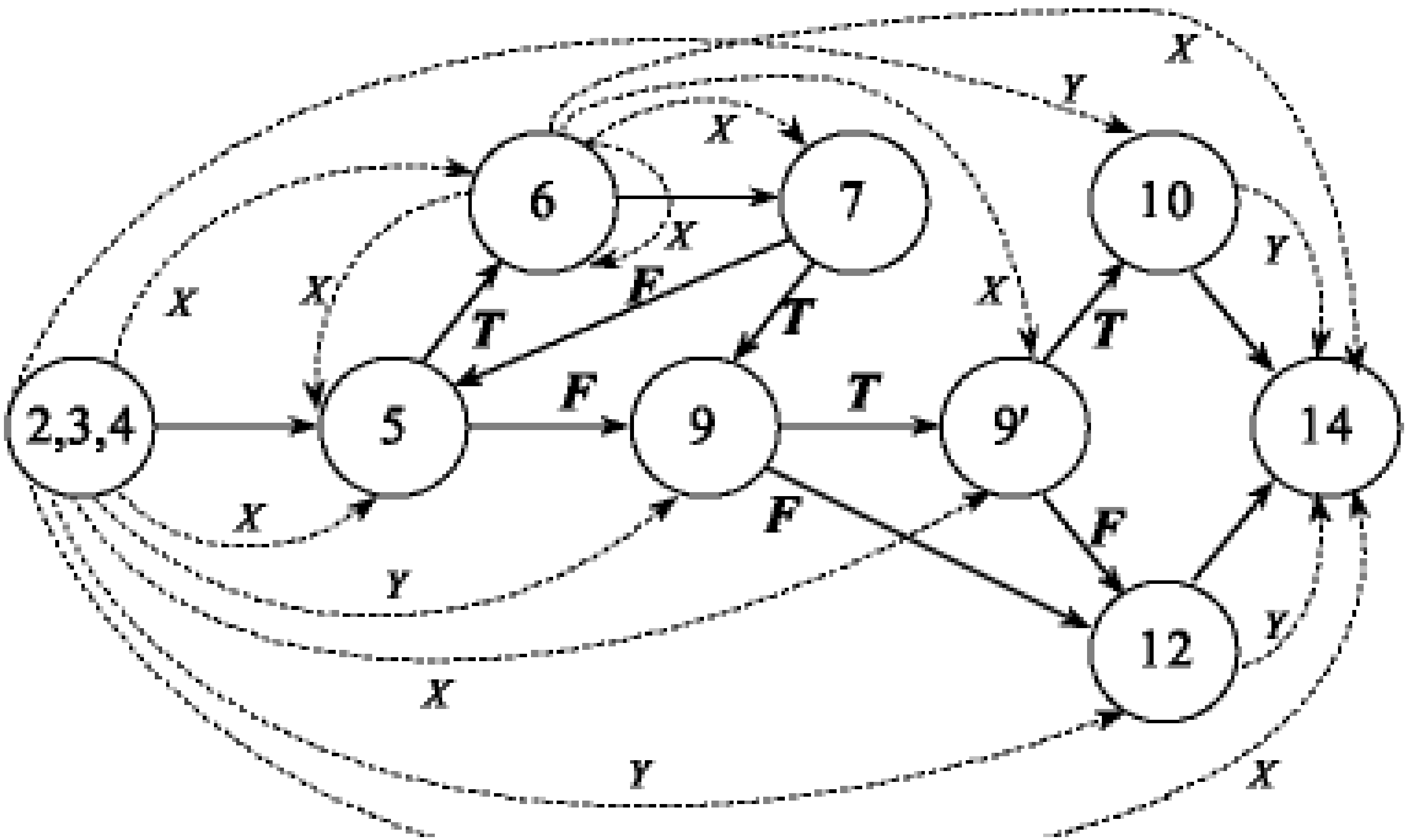


Example all-edges-adequate test set:
(X = 20, Y = 10)
(X = 5, Y = 30)
(X = 21, Y = 10)

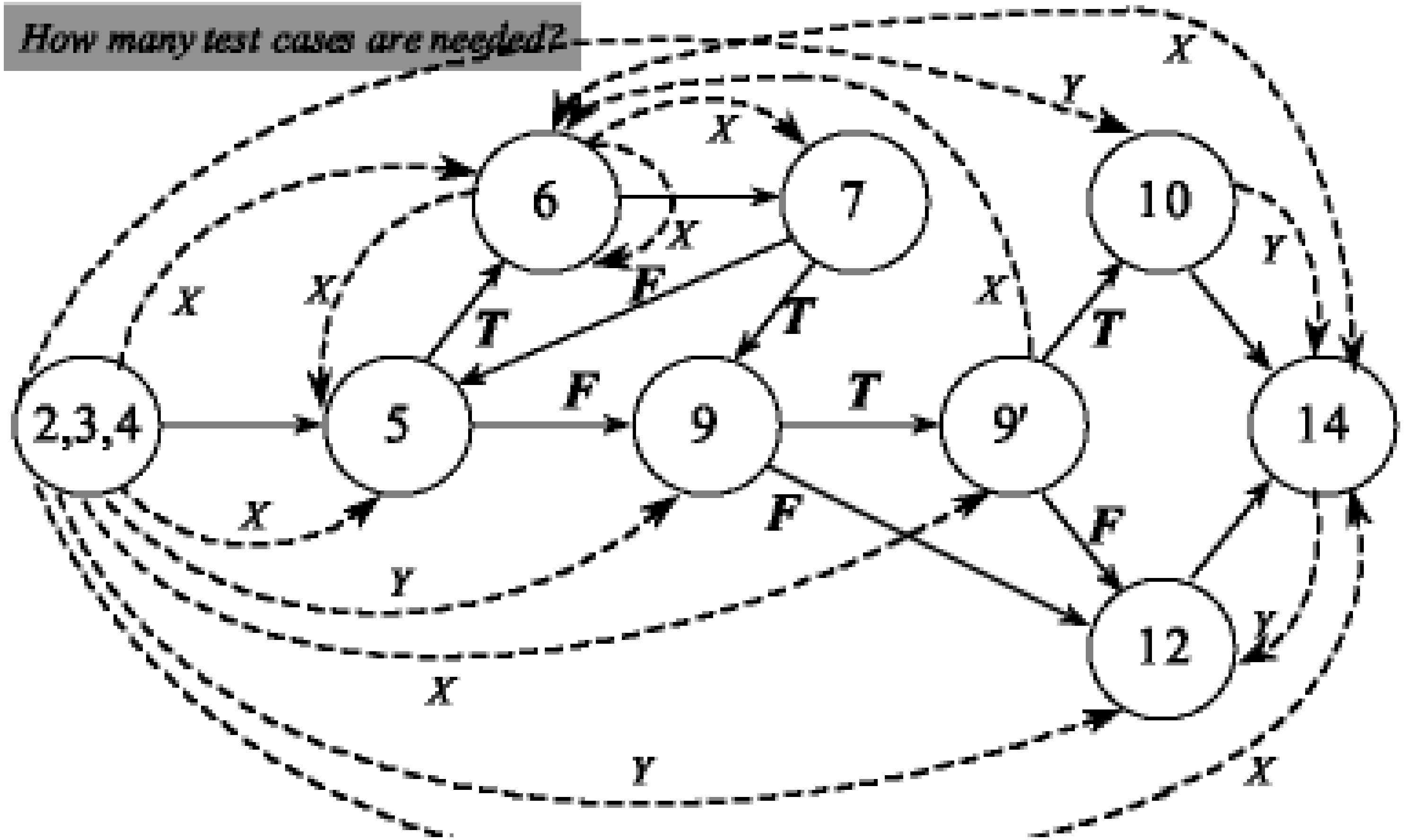
P's CFG with a Data Flow Edge



P's Control and Data Flow Graph



All–Uses Coverage of P

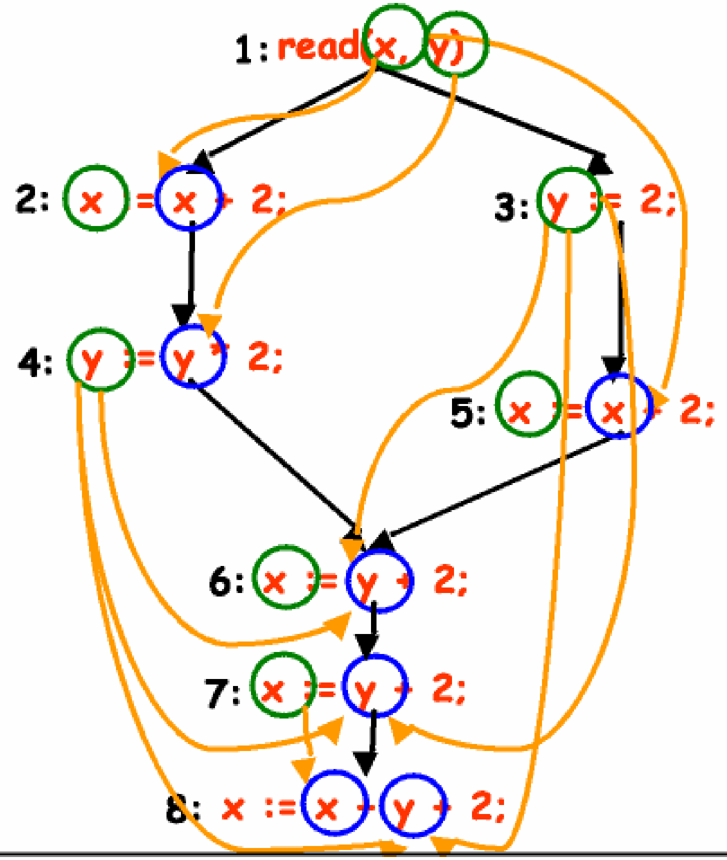


Structural Testing

– Data-flow based adequacy criteria

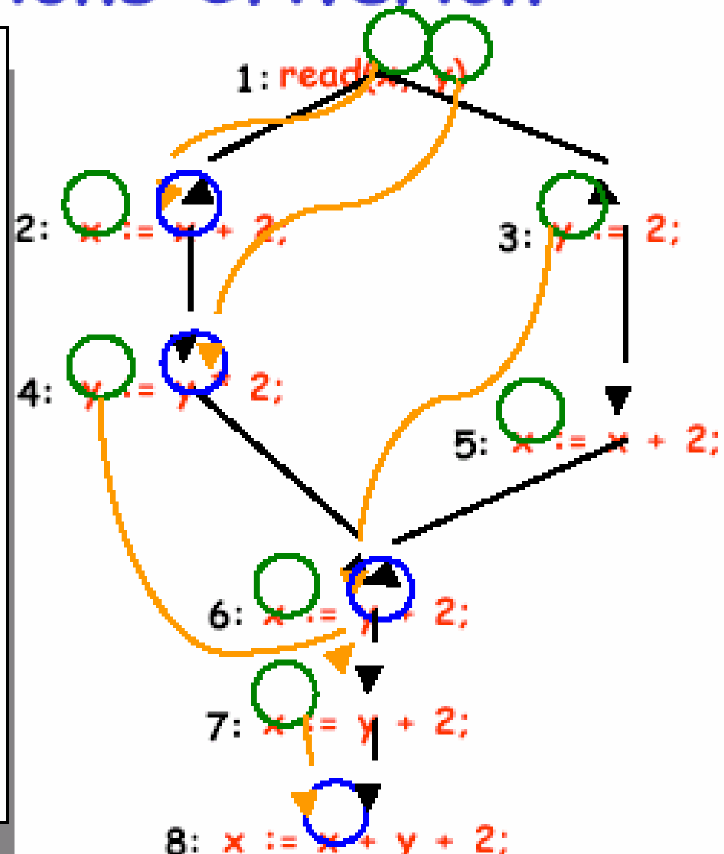
- All definitions criterion
  - Each definition to some reachable use
- All uses criterion
  - Definition to each reachable use
- All def-use criterion
  - Each definition to each reachable use

Data-flow Testing

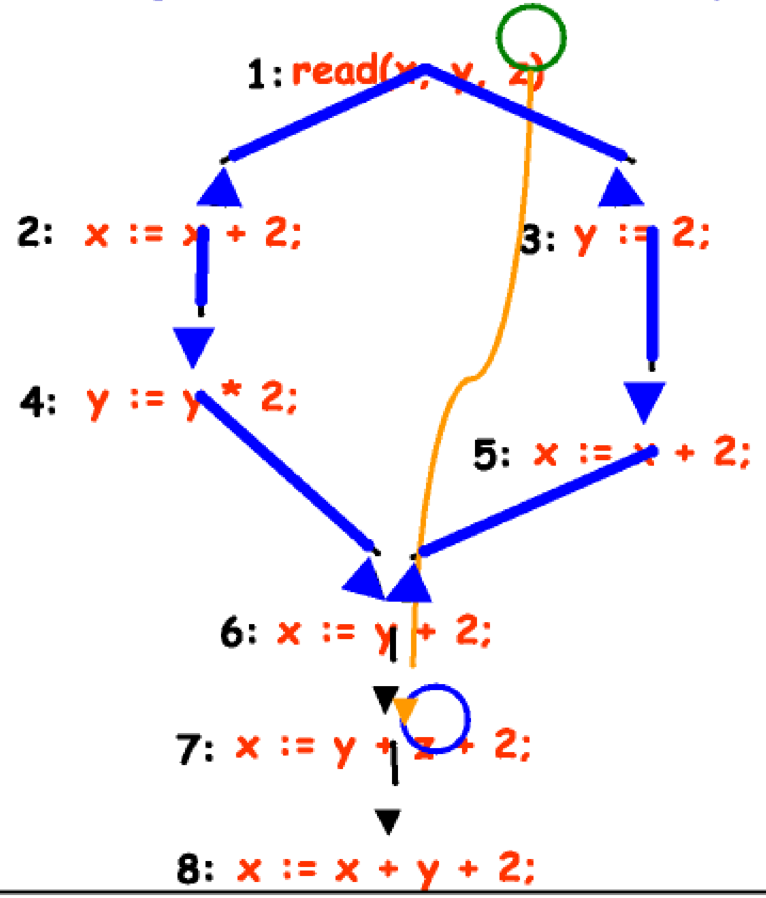


All Definitions Criterion

- A set P of execution paths satisfies the all-definitions criterion iff
  - for all definition occurrences of a variable x such that
    - there is a use of x, which is feasibly reachable from that definition,
  - there is at least one path p in P such that
    - p includes a subpath through which the definition of x reaches some use occurrence of x



All Uses Criterion



All DU-paths criterion

- A set P of execution paths satisfies the all-DU paths criterion iff
  - for all definitions of a variable x and all paths q through which that definition reaches a use of x,
  - there is at least one path p in P such that
- q is a subpath of p and q is cycle-free

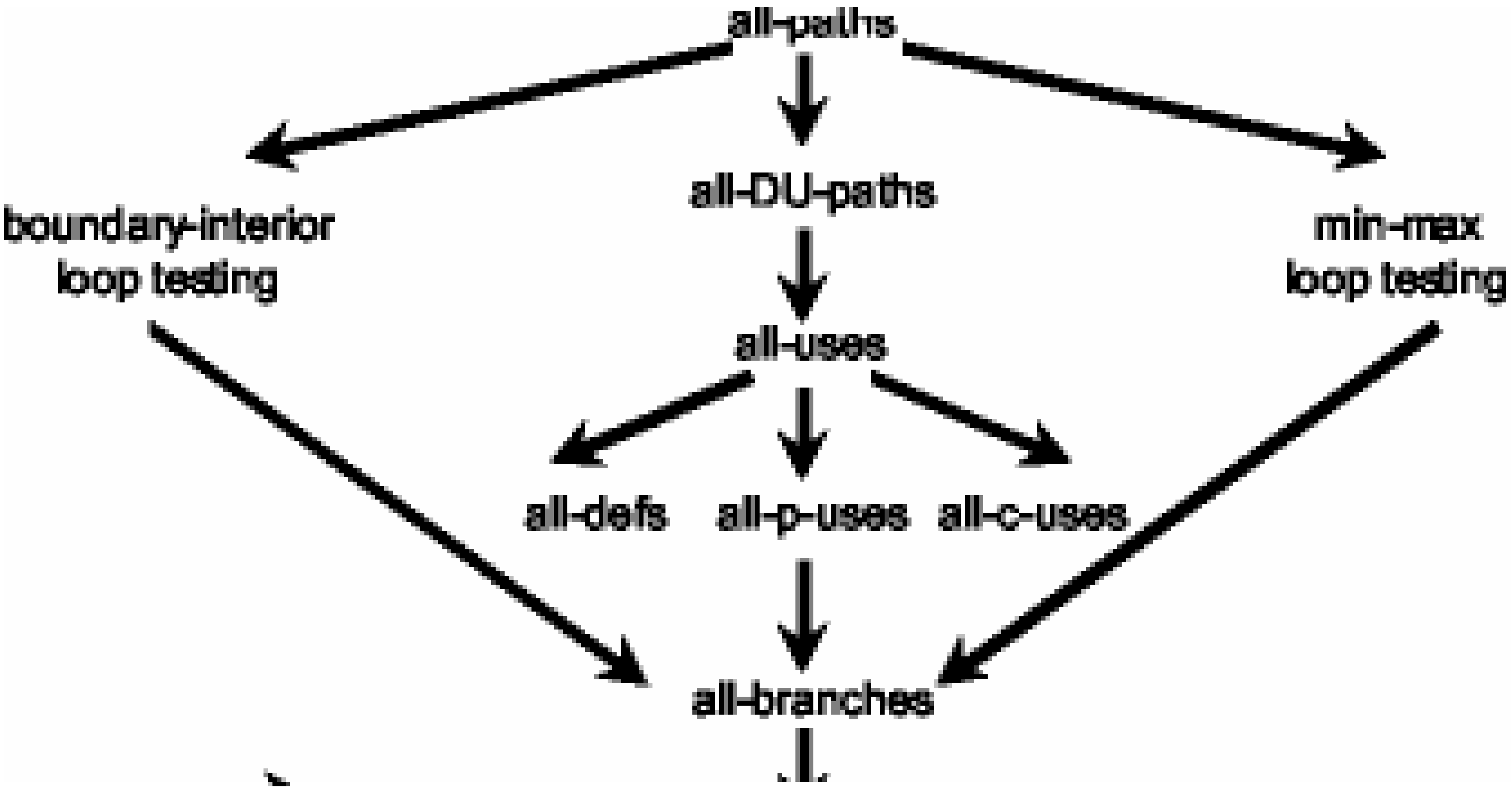
Subsumption

- Criteria C1 subsumes criteria C2, iff
  - For all programs p being tested with specifications s
  - All test sets t
  - t is adequate according to C1 fortesting p with respect to s impliesthat t is adequate according to C2 fortesting p with respect to s
- Path subsumes branch
- Path subsumes statement

Subsumption and Covers

- C1 *subsumes* C2 if any C1-adequate T is also C2-adequate – But some T1 satisfying C1 may detect fewer faults than some T2 satisfying C2
- C1 *properly covers* C2 if each subdomain induced by C2 is a union of subdomains induced by C1

Clarke, Podgurski, Richardson & Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE Transactions on Software Engineering, November 1989.


all-paths
all-DU-paths
boundary-interior
loop testing
min-max
loop testing
all-uses
all-defs
all-p-uses
all-c-uses
all-branches

Challenges in Structural Coverage

Interprocedural and gross-level coverage

- **e.g., interprocedural data flow, call-graph coverage**

Regression testing

Late binding (OO programming languages)

- **coverage of actual and apparent polymorphism**

Fundamental challenge: Infeasible behaviors

- **underlies problems in inter-procedural and polymorphic coverage, as well as obstacles to adoption of more sophisticated coverage criteria and dependence analysis**

The Infeasibility Problem

- Syntactically indicated behaviors (paths, data flows, etc.) are often impossible
  - Infeasible control flow, data flow, and data states
- Adequacy criteria are typically impossible to satisfy
- Unsatisfactory approaches:
  - Manual justification for omitting each impossible test case (esp.. for more demanding criteria)
  - Adequacy “scores” based on coverage

    example: 95% statement coverage, 80% def-use coverage

Coverage and Components

State and Encapsulation

- **Procedural programming**
  - **– Basic component: Subroutine**
  - **– Testing method: Subroutine input/output based**
- **Object-oriented and component programming**
  - **– Basic component: Class = Data structure + Set of operations**
  - **– Objects are instances of classes**
  - **– The data structure defines the *state* of the object. Correctness is not based only on output, but also on the state.**
  - **– The data structure is not directly accessible, but can only be accessed using the class *public* operations *(Encapsulation)*.**
- **Problems:**
  - **– What are the basic elements to test?**
  - **– Is it enough to observe input/output relations?**
  - **– How is it possible to observe the state without violating encapsulation?**
  - **– What if the source code is not available (for a third-party component)?**